AFRL-HE-WP-TP-2007-0008

2002 Defense Modeling and Simulation Office (DMSO) Laboratory for Human Behavior Model Interchange Standards

Michael van Lent
Randall Hill
Ryan McAlinden
Paul Brobst

University of Southern California
Institute for Creative Technologies
13274 Fiji Way
Marina del Rey CA 90292

July 2003

Interim Report for August 2002 – July 2003

Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
Wright-Patterson AFB OH 45433-7604

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* July 2003	**2. REPORT TYPE** Interim	**3. DATES COVERED** *(From - To)* August 2002 - July 2003

4. TITLE AND SUBTITLE
2002 Defense Modeling and Simulation Office (DMSO) Laboratory for Human Behavior Model Interchange Standards

5a. CONTRACT NUMBER NAWC-TSD-BAA-2.3.2

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER 63832D

6. AUTHOR(S)
Michael van Lent, Randall Hill, Ryan McAlinden, Paul Brobst

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER 0476DM00

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Institute for Creative Technologies
13274 Fiji Way
Marina del Rey CA 90292

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Materiel Command
Air Force Research Laboratory
Human Effectiveness Directorate
Warfighter Interface Division
Cognitive Systems Branch
Wright-Patterson AFB OH 45433-7604

Defense Modeling and Simulation Office
1901 N. Beauregard Street, Suite 500
Alexandria VA 22311-1705

10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/HECS, DMSO

11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-HE-WP-TP-2007-0008

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.
AFRL/PA cleared on 26 June 2007, AFRL-WS-07-1525.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the effort to address the following research objective: "To begin to define, prototype, and demonstrate an interchange standard among Human Behavior Modeling(HBM)-related models in the Department of Defense (DoD), Industry, Academia, and other Government simulations by establishing a Laboratory for the Study of Human Behavior Representation Interchange Standard." With experience, expertise, and technologies of the commercial computer game industry, the academic research community, and DoD simulation developers, the Institute for Creative Technologies discusses their design and implementation for a prototype HBM interface standard and also describes their demonstration of that standard in a game-based simulation environment that combines HBM models from the entertainment industry and academic researchers.

15. SUBJECT TERMS Human Behavior Modeling, Unreal Tournament, Human Behavior Modeling Interchange Architecture, Interchange Protocol

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			John L. Camp
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR	54	**19b. TELEPHONE NUMBER** *(include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. 239.18

THIS PAGE LEFT INTENTIONALLY BLANK

Table of Contents

THIS PAGE LEFT INTENTIONALLY BLANK

1.0 Introduction

The research objective of the 2002 Laboratory for Human Behavior Representation Interchange Standards effort has been: "To begin to define, prototype, and demonstrate an interchange standard among HBM-related models in DoD, Industry, Academia, and other Government simulations by establishing a Laboratory for the Study of Human Behavior Representation Interchange Standard, addressing research needs in NAWC-TSD BAA 2.3.2 ("Advanced Human Behavioral Representation Techniques")." To address this research objective, researchers at the University of Southern California's Institute for Creative Technologies have drawn on the experience, expertise and technologies of the commercial computer game industry, the academic research community, and DoD simulation developers. With these three communities ICT has designed a prototype HBM interface standard and demonstrated that standard in a game-based simulation environment that combines HBM models from the entertainment industry and academic researchers.

1.1 Project Goals

As defined in the proposal and statement of work the 2002 Laboratory for Human Behavior Representation Interchange Standards set out to accomplish three major tasks:

- The first task was to "Design, integrate, and document a laboratory capable of running a modest (i.e., several) collection of CGFs (Computer Generated Forces) from DoD (Department of Defense) and academia, and NPCs (Non-Playing Characters) from EI (Entertainment Industry) and academia that exchange data and control according to the evolving HBM (Human Behavior Model) interchange standard." The Defense Modeling and Simulation Office (DMSO) HBM Lab contains four desktop systems and two laptops, all Windows/Linux dual boot, as well as a full suite of development software. This lab has been set up at ICT in Marina del Rey, CA and Section 2 of this final report documents the laboratory facilities. This lab includes three HBMs, Soar and PMFServ from academia and AI.Implant from the entertainment industry, as well as two simulation environments, Unreal Tournament from the entertainment industry and OneSAF TestBed from the DoD.
- The second task was to "Examine the HBM (Human Behavior Model)-related attributes within each relevant CGF and NPC instance, then coordinate and document the specification of these elements with the HSTC (HBM Standards Technical Committee)." As the HSTC was never organized ICT, in collaboration with other DMSO-selected research groups, undertook the design, coordination, and documentation of a prototype Human Behavior Representation Interface Standard. The standard defines both a multi-threaded control scheme and an initial set of data elements (sensor inputs and control output) as described in Section 3 of this final report. A related sub-task is to "Identify the HBM-related elements of the task that require standardization." As described in Section 6.3, spatial representation is an area that requires additional attention relating to standardization.

- The third task was to "Collaborate with DMSO, designated contractors, and other DMSO-sponsored laboratories in developing one or more scenarios derived from both EI and DoD domains to be the basis for this year's experimentation." The Blackhawk Down-inspired, asymmetric, urban combat scenario, developed in conjunction with DMSO, the University of Pennsylvania PMF (Performance Moderator Function) team, Quicksilver software, and the AI.Implant team from BioGraphic Technologies is this demonstration scenario. For the first time this scenario demonstrates three different human behavior models integrated into a single virtual environment at the same time. In addition, these HBMs control NPCs playing different roles (U.S. Army Ranger, civilian crowd, asymmetric opponent) through slight variations of a single interface. The HBMs and simulation environment are described in Section 4 and the demonstration scenario is described in Section 5.

In addition to the sections already mentioned that describe how these three major tasks were accomplished Section 6 presents some opportunities for future work. Finally, Section 7 details when and how each deliverable defined in the statement of work was satisfied.

2.0 <u>DMSO HBM Lab</u>

This section details the design for the laboratory for Human Behavior Representation Interchange Standards at the University of Southern California's Institute for Creative Technologies. The laboratory design consists of a description of the physical space the lab will occupy, the computers the lab will contain, and the software that will be supported. Based on this design the lab was completed in April, 2003.

<u>Lab Space</u>

The laboratory for Human Behavior Representation Interchange Standards occupies room 1115 on the 11th floor of the North Tower building in Marina del Rey CA 90292. Room 1115 contains five desks and desk chairs. In addition to the laboratory space there is also be a programmer's office located next door in room 1130.

<u>Hardware</u>

The laboratory for Human Behavior Representation Interchange Standards consists of seven computers; four desktop PCs, two notebook PCs, and a single, rack mounted version management server. The four desktop PCs serve as the primary development machines and support in house demonstrations. The two notebook PCs serve as the primary demonstration machines for demonstrations at other facilities. Both the desktop and notebook PCs are capable of running multiple simulation environments (OTB, DISAF, Unreal Tournament), multiple human behavior model (HBM) components, and are HLA capable. The exact specifications for the desktop PCs and notebook PCs are listed below.

The ICT version management server is designed to support all the ongoing software development projects at ICT. The laboratory for Human Behavior Representation Interchange Standards makes use of this server to manage multiple versions of the both

HBM software developed at ICT and the software provided by outside collaborators for integration and testing at ICT. In addition, the ICT also has an FTP server, a SourceForge server, and a Microsoft Exchange server in place, all of which are utilized by the laboratory for Human Behavior Representation Interchange Standards.

Desktop PCs

Four Dell Dimension 8200 desktop PCs (at $2,619.48 each)

- Pentium 4 Processor at 2.8GHz
- 1GB PC800 RDRAM
- 80GB Ultra ATA/100 7200 RPM Hard Drive
- 18.1 inch Flat Panel Display
- 64MB nVIDIA GeForce4 Ti 4200 Graphics Card
- Sound Blaster Live! Sound Card
- Harman Kardon HK-206 Speakers
- CD-RW/DVD Combo Drive
- 3.5 inch Floppy Drive
- 10/100 Fast Ethernet Card
- Keyboard and Optical Mouse

Notebook PCs

Two Dell Inspiron 8200 notebook PCs (at $3,662.76 each)

- Mobile Pentium 4 Processor at 2.2GHz
- 15.0 inch UltraSharp display
- 1GB DDR RAM
- 40GB Ultra ATA 5400RPM Hard Drive
- 3.5 inch Floppy Drive
- CD-RW/DVD Combo Drive
- 64MB nVIDIA GeForce4 440 Go Graphics Card
- Integrated Network Card and Modem
- Internal TrueMobile Wireless Network Card
- Nylon Carrying Case

Software

To be capable of running the multiple simulation environments required the desktop and notebook PCs support two operating systems; Microsoft Windows 2000 and Linux. All the PCs are dual boot, allowing the user to specify which operating system to use when the computer is powered up. In addition to the operating systems the PCs are also loaded with development environment software, simulation environment software, and a variety of HBM component software.

Operating Systems

The desktop and notebook PCs support both Microsoft Windows 2000 and Linux in a dual boot configuration. Our experience has been that Windows 2000 is more stable and more secure that Windows XP at the present time. The Windows 2000 operating system was delivered with the desktop and notebook PCs as part of the cost of those systems. If a switch is necessary at a later time all PCs will be capable of supporting Windows XP also.

Development Environments

For development in Windows 2000 the desktop and notebook PCs are loaded with Microsoft Visual C++ 6.0. ICT currently has a site license for Visual C++ 6.0 that covers the Human Behavior Representation Interchange Standards laboratory. Development in Linux primarily relies on the gcc compiler and a variety of editors (emacs, vi) all of which are freely available as part of the Linux distributions. The ICT version management server runs the open Concurrent Version System (CVS) software.

Simulation Environments

Initially the laboratory for Human Behavior Representation Interchange Standards supports three simulation environments. Unreal Tournament is a commercial computer game published in 2001 by Epic Games Inc. Although Unreal Tournament is primarily entertainment based it can also serve as a reasonably complex simulation environment for small unit (squad-level) tactical combat. This is especially true with the addition of the Infiltration mod that modifies the game setting and simulation rules into a much more realistic tactical squad-level combat simulator. Unreal Tournament with the Infiltration mod is supported on the desktop and notebook PCs. Six copies of Unreal Tournament have been purchased for $20.00 each. The Infiltration mod is available for download free of charge.

In addition, OneSAF Testbed has been installed in three additional machines (funded by a non-DMSO project) that reside in the lab space. In addition to OneSAF Testbed, these machines also have MPARS (the Mission Planning and Rehearsal Exercise). Finally, a binary release of DISAF version 9.4 has been installed on the desktop machines in the DMSO lab.

Human behavior modeling components

The primary focus of the Human Behavior Representation Interchange Standards project is to develop interchange standards between a variety of HBM components the laboratory also supports a number of these components.

The Soar architecture is "a unified architecture for developing intelligent systems". Soar will be used to control the human user's subordinates in the demonstration scenario. Soar will be installed on all the desktop and notebook PCs and will run in both the Windows 2000 and Linux operating systems. ICT has also obtained the Soar General Input Output (SGIO) interface between Soar and Unreal Tournament:Infiltration which will run in the Windows 2000 operating system. The Soar architecture software is available for download free of charge.

AI.Implant is a HBM component from the commercial game development industry that focuses on navigation and path planning in a complex, dynamic environment. AI.Implant is a middleware system developed by BioGraphic Technologies, Inc. AI.Implant will control the opponent militia in the demonstration scenario. BioGraphic Tech. has provided ICT with a free license for the AI.Implant SDK for Windows 2000 for the duration of this project. AI.Implant has been installed on desktop and notebook PCs and runs in Windows 2000.

Performance Moderator Functions (PMFs) are software packages that moderate the behavior of an HBM based on physiological factors (such as fatigue, pain) and emotional factors (such as fear, anger, panic) among others. PMFServ is a central architecture for these PMFs that includes a simple behavior generation component. PMFServ will control civilians, crowds and some opponent forces in the demonstration scenario. The laboratory supports PMFServ in the Windows 2000 operating system.

Schedule for Laboratory Construction

Approval to begin purchasing equipment for the laboratory was obtained in February, 2003. Construction proceeded as follows:

February 2003

Orders placed for desktops and first notebook PCs.
Furniture moved into laboratory and office space.

March 2003

Desktops and first notebook arrive and are installed.
Begin installing operating systems and software.
Final notebook ordered.

April 2003

All software installed and laboratory fully functional

3.0 Prototype HBM Interface Standard

The Prototype Interchange Standard that was developed as part of this initiative is an interface that allows disparate Human Behavior Models (HBMs), in this case Soar, AI.Implant and PMFServ, to control entities in a single simulation environment, in this case Unreal Tournament. The unique advance demonstrated by this project is the use of a single interface to support three HBMs running concurrently within a simulation[1]. In essence, the HBMs are three uniquely developed external software modules that operate asynchronously through an interface with the environment. The Interchange Standard is a control methodology and set of data specifications that was developed to allow for the integration of a wide variety of HBMs into multiple simulation environments. A general overview of each HBM is provided below with an in-depth description of interface to follow.

[1] To describe this as it relates to Unreal Tournament, there are a variety of non-player characters (NPCs) within the game that are controlled by each of these HBMs interacting with the game engine.

- Soar: Initially developed at Carnegie Mellon University by Allen Newell, John Laird, and Paul Rosenbloom, this is the HBM that will control the Non-Player Character (NPC) movement, formation, and attack behaviors for the U.S. Army Rangers working as subordinates of the human user.
- AI.Implant (AII): Developed by Biographic Technologies, this AI middleware tool focuses on path-planning and navigation to control NPC movement of opponent militia in the demonstration scenario.
- Performance Moderator Functions (PMFs): Developed by Barry Silverman and team at the University of Pennsylvania, PMFs are individual components that model emotional and physiological effects such as stress, perception, and emotional utility. PMFs are integrated in the PMFServ architecture. In the demonstration scenario PMFs are designed to control civilians and crowd members.

As reflected above, each of the HBMs demonstrate a different aspect of human behavior. As discussed in Section 6.0, an interesting direction for future work is to integrate these HBMs with each other to create a combined human behavior model. For example, the Interchange Standard could support an NPC that has his high-level behavior determined by Soar, low-level path planning and navigation controlled by AI.Implant, and emotional and physiological states managed by the PMFs.

The most important aspect of the prototype HBM Interface Standard is that the standard is independent of any specific HBM or simulation environment. Even though some of the interface code developed for this project is designed for Unreal Tournament, the methodology used is quite broad. Simulation data are polled each iteration through the simulation's tick cycle, distributed to the appropriate HBM, processed, and a request is returned to the simulation for execution (move, attack, orient). The actual software written for this project may change depending upon the simulation environment used, but the structure and concepts employed would not.

3.1 Design Approach

The Prototype HBM Interface Standard (HBM IS) is the codification of a great deal of previous experience with HBM design on the part of a number of members of the development team. As a starting point the HBM IS builds off the Soar General Input/Output (SGIO) interface to Unreal Tournament developed at the University of Michigan. This interface is the fifth HBM interface developed by the University of Michigan Soar group. Previous interfaces include the TacAir Soar interface to ModSAF, as well as interfaces to the SGI Flight Simulator, Quake II, Descent III, and FreeCivilization. Dr. van Lent participated in the development of three of these previous interfaces as well as the interface used by the virtual humans in the Mission Rehearsal Exercise at ICT.

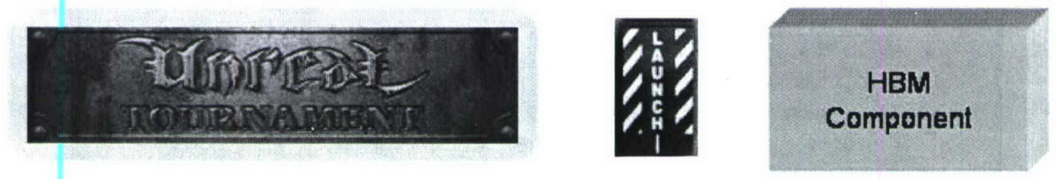
While these previous interfaces have covered a range of simulation environments they have all been designed for the Soar HBM. To ensure coverage of other HBMs the interface design team included the experience of the Performance Moderator Function team at the University of Pennsylvania and the AI.Implant developers from BioGraphics

Technologies. The starting HBM IS was extended and modified to ensure compatibility with these two additional HBMs.

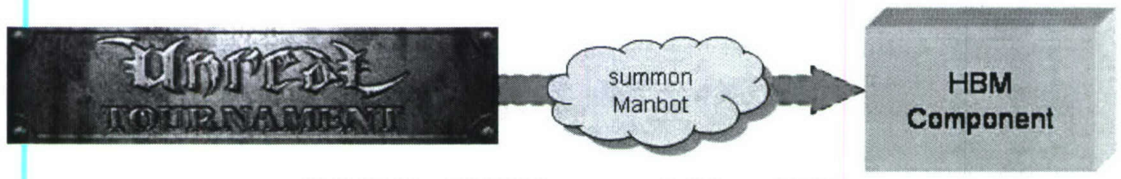
The focus of the current prototype HBM IS is control of dismounted soldiers in urban environments. Future extensions to the HBM IS should seek to expand the scope to include control of a broader range of platforms (main battle tanks, infantry fighting vehicles, fixed and rotary wing aircraft, ships…) as well as control of higher level command entities (squads, platoons, companies…).

3.2 Control

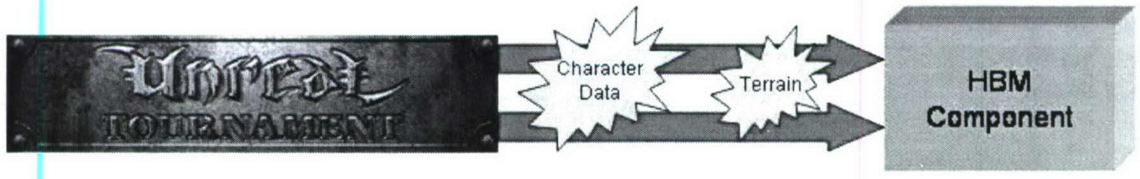
The quantity of simulation-related data coming into each of the HBMs through the HBM IS is extremely large. Flow of data from the simulation occurs multiple times per second (generally about 10 times/second), requiring the input mechanism for the HBM to constantly be monitoring for the most recent simulation updates. Below are a set of illustrations that depict the way data is routed through the UT/HBM Architecture. A brief description follows each diagram.



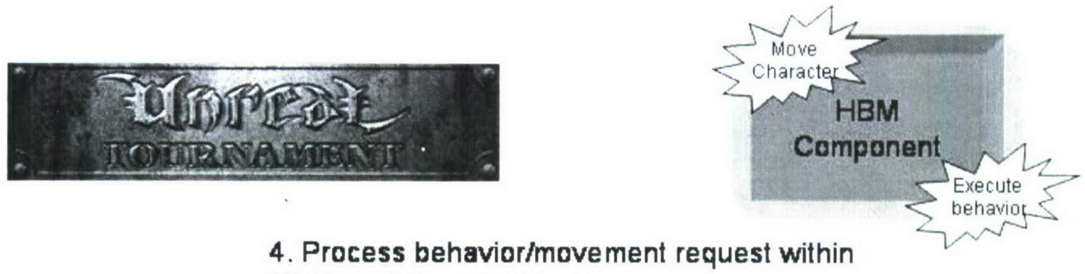
1. Initialize Unreal Tournament (Start of Game)– First Tick() is called



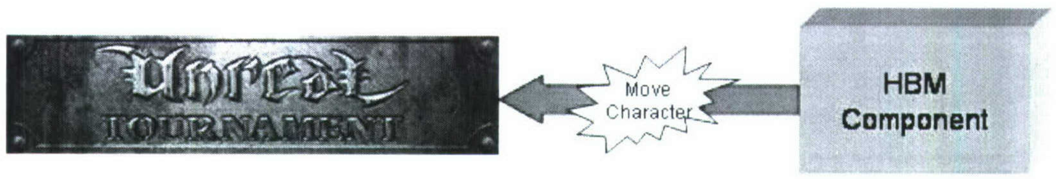
2. Initialize HBM Component (Soar, All) by instantiating the DLL through Unreal Tournament



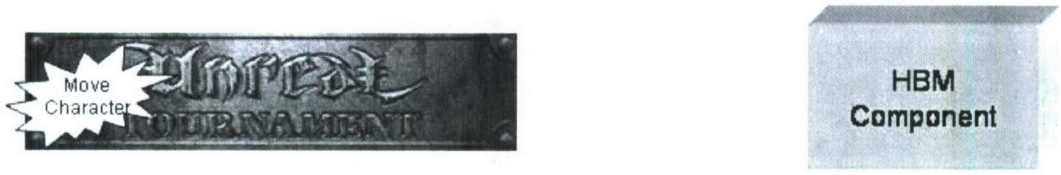
3. HBM requests the retrieval of environment information from Unreal Tournament



4. Process behavior/movement request within HBM modules (All, Soar)



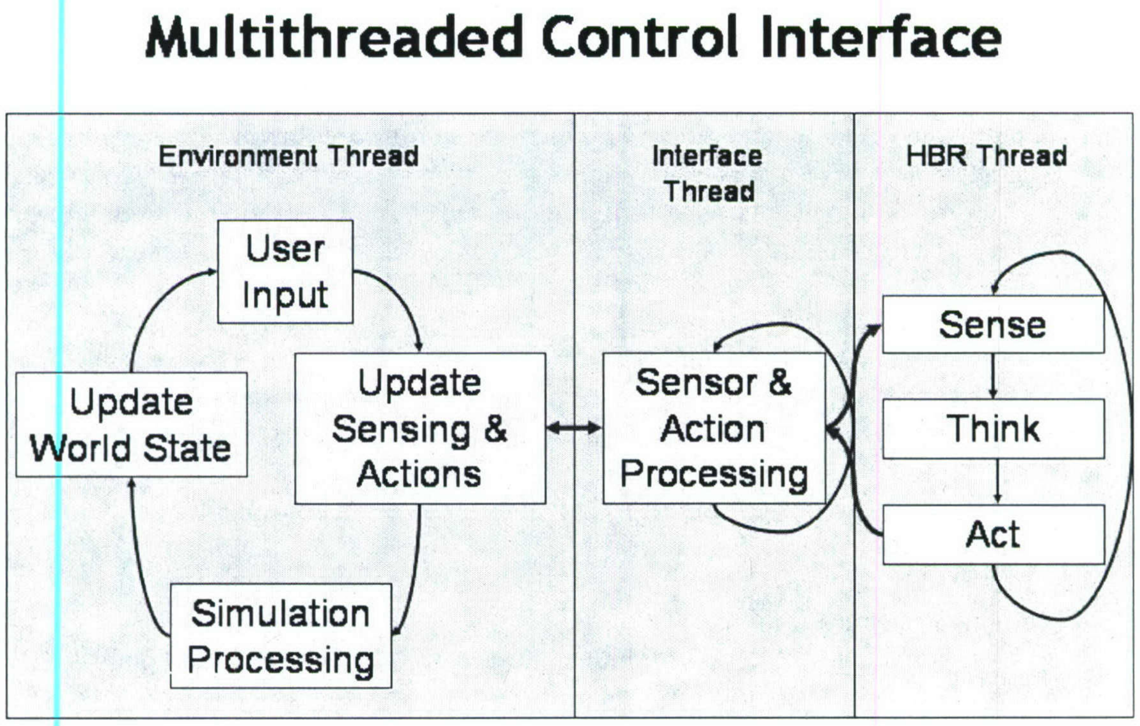
5. Return command to Unreal Tournament for execution



6. Execute command/behavior and iterate through Tick() once again

These diagrams describe the flow of data between the simulation environment and HBM components. Note that the "HMB Component" box represents Soar, AI.Implant or PMFServ. The communication mechanism for the exchange of data integrated into a Dynamically Linked Library (DLL) that is loaded into the game engine process during execution. There is no socket connection or shared memory processes. As a result, the interfaces between the simulation and HBM components are tightly coupled, which improves efficiency but decreases flexibility when looking to deploy in other environments (such as a Java-based simulation) because DLLs are a Microsoft Windows specific mechanism. The SGIO interface supports both a DLL interface and socket-based interface. The differences between the two interface styles are completely contained in the SGIO system and neither the HBM or simulation environment changes when switching between the DLL and socket versions.

One interesting aspect in the development of the interface was the threading techniques used. The Unreal Engine is, by design, intended to run within a single thread for game efficiency. The HBM interface, however, uses a multithreaded approach to allow the simulation environment and HBM component to run asynchronously. This allows both Unreal Tournament and the HBM to run without impacting each other's execution loops. Since the UT engine runs as a single thread each step in the internal game loop must be carefully controlled to execute and return quickly. If any step takes too long the frame rate of the game can drop below the acceptable 30 frames a second or "hiccup" by freezing on a single frame for too long while the engine waits for a step in the loop to finish. In all game engines a great deal of effort is required to ensure that each step in the game loop returns quickly and consistently. HBMs, especially those from the academic research community, are not engineered to fit into these strict time limits. Soar, for example, generally takes less than 50 msec per cycle but in degenerate cases may take seconds per cycle. Encapsulating the HBM in a separate processing thread allows the game engine and HBM component to operate independently.



It is important to note that the Interface and HBM Threads are independent of model type; they all use the same methodology for receiving/outputting information from/to Unreal Tournament. Essentially, what is occurring each iteration through the game loop is as follows:

- Unreal Tournament maintains information on the state of the "world," which includes entity locations, health, etc.
- Each HBM, operating within its own thread, polls for this information and uses it as input in its decision cycle
- Once a decision (command) has been issued by the HBM, it is sent back over the interface thread to Unreal Tournament for execution. If the information is not sent quick enough (i.e. before the game loop ends), it is discarded and the process repeats itself
- The command (move-to, attack, etc.) is loaded into the Engine and executed

3.3 Data

In order for the HBMs to accurately model any level of intelligent behavior, path-planning, or emotional effects, sensor data must be pulled from the simulation to serve as input to the particular behavior model. This sensor data comprises a wide variety of information about the environmental. Such data includes:

- Player and Character information (name, position, team, equipment, health status)
- Static and dynamic objects and terrain in the game (obstacles, doors, projectiles)
- General mission information (map name, game type)
- Spatial representations (path node locations)
- Player/Character communications (formation, engage requests)

There are a variety of mechanisms for exchanging this information between the simulation and HBM including sockets, shared memory, shared file access, remote procedure calls, and dynamically loadable libraries (DLLs). Each of these alternatives has advantages and disadvantages some of which are summarized below:

- Socket:
    - Advantages: Fairly platform independent, Allows communication between machines.
    - Disadvantages: Low bandwidth, significant latency
- Shared Memory:
    - Advantages: High bandwidth
    - Disadvantages: Platform specific, no communication between machines, some latency
- Shared File Access:
    - Advantages: Very platform independent, Allows communication between machines
    - Disadvantages: Very slow, Lots of latency
- Remote Procedure Calls:
    - Advantages: Allows communication between machines, Fair bandwidth
    - Disadvantages: Platform specific, significant latency
- DLLs:
    - Advantages: High bandwidth, very low latency, builds on existing game industry interfaces
    - Disadvantages: No communication between machines, Platform specific

The Human Behavior Model Interchange Standard currently uses the DLL mechanism for exchanging information with the simulation environment. A DLL-based interface was chosen for the advantages listed and because it is the main interface mechanism used by the commercial game industry. Another approach, taken by the Soar General Input Output (SGIO), is to support one mechanism (Sockets) for development and debugging and a second mechanism (DLLs) for the final system.

Through the DLL interface the HBM IS supports a wide variety of different input sensors and action outputs. These sensors and actions are described in more detail later in this section. Each of the specific HBMs used in this project takes advantage of a subset of the full sensor and action suite. In two cases (AI.Implant and PMFServ) the sensors and actions are filtered on the simulation side to reduce the amount of data transmitted. Soar processes all sensor inputs and has all actions available although not all of this data is used in the current demonstration. The primary data structures polled from the environment and used by each of the HBMs are as follows:

- Soar:
  - Game Items/decorations
  - Projectiles
  - Entities
  - NPC Attributes: physiology, items, weapon, feelers
  - Game Attributes
  - Map Information
  - Sound
  - Feedback
  - Player Messages
- AI.Implant:
  - Player attributes
  - Entities
  - NPC Attributes
- PMFs

The HBM IS data interface is organized as a multi-level hierarchy. The top level of the hierarchy groups the sensors into general classes; agent, feedback, objects, game, map, sound, message. The agent sensors poll information about the specific character controlled by the HBM. The feedback sensors give the HBM propreoceptive feedback about the status of the various actions available. The objects sensors report on any other objects in the environment including other characters, items (such as weapons), and projectiles. The game sensors give details about the specific parameters of the current game. Map sensors give information about the static terrain including any invisible waypoint nodes and distances to barriers in eight directions. Sound sensors report on any audio elements of the environment including footsteps and weapon firing from other characters. Finally, message sensors are used to communicate between characters and with the user. The full set of prototype Human Behavior Model Interchange Standard sensors is listed here with a brief description of each sensor.

<u>Prototype Data Interface</u>

<u>Sensor Input</u>

Agent

```
angle: direction/angle the character's body is facing
       pitch(float): vertical angle (0=straight down)
       yaw(float): horiztonal angle (0=north)
       roll(float): body rotation (0=upright)
armor-amount(int): amount of UT armor the character has
cycle(int): HBM cycle count
fatigue(float): character's fatigue level
health(int): amount of UT health the character has (0-100)
hunger(float): character's hunger level
light-level(int): amount of light in immediate environment
name(string): character's name
pain(float): character's pain level
fear-factor(int): character's fear level
position: current position of character's body
```

```
      x(float)
      y(float)
      z(float)
random(float): random number updated each cycle
strength(float): character's strength level
team(string): name of character's team
area-temp(float): temperature of immediate surroundings
body-temp(float): character's body temperature
thirst(float): character's thirst level
time(float): current wall clock time
velocity: velocity of character's body
      x(float)
      y(float)
      z(float)
posture: (kneel/kneeling/prone/going-prone/stand/standing)
item: record of each item character is carrying
      name(string): name of item
      class(string): type of item
      quantity(int): how many are being carried
      selected(yes/no): currently held or in "pack"
      in-use(yes/no): currently being used
weapon: record of each item character is carrying
      name(string): name of weapon
      ammo-type(string): type of ammo weapon uses
      ammo-amount(int): amount of ammo currently in weapon
      selected(yes/no): currently held or in "pack"
```

Feedback

```
move-target(off/on): is move to target currently enabled
lead-target (off/on): is lead target currently enabled
face-target(off/on) : is face target currently enabled
attack (normal/alt/charge/charge-alt/none): currently
attacking?
thrust (back/front/none): moving forward or backward?
side-step(left/right/none): moving sideways?
turn(back/front/left/right/none): turning?
```

Objects

```
Item: a dynamic item in the environment
      name(string): name of item
      range(float): distance to item from current position
      angle-off: angle item is off from character facing
            h(float): horizontal angle-off
            v(float): vertical angle-off
      position: global position of item
            x(float)
            y(float)
            z(float)
node: a visible pathing node in the environment
      range (float): distance to node
      position: global position of node
            x(float)
            y(float)
            z(float)
      angle-off: angle node is off from current facing
            h(float)
            v(float)
      name(string): name of node
      connect-heading(float): heading to move to next node
      is-door(yes/no): is this a special door node?
      is-window(yes/no): is this a special window node?
projectile: a projectile (bullet) in the environment
      name(string): name of projectile type
      range(float): distance to projectile
      angle-off angle proj. is off from character facing
            h(float)
            v(float)
      position: global position of projectile
            x(float)
            y(float)
            z(float)
      velocity: velocity of projectile in 3 dimensions
            x(float)
            y(float)
            z(float)
entity: another character in the environment
      name(string): name of character
      health(int): character's health level (0-100)
      weapon(string): weapon character is holding
      visible(true/false): am I visible to the character?
      team(string): character's team
      angle-off: angle char. is off from my facing
            h(float)
            v(float)
      aspect : angle I am off from character's facing
            h(float)
            v(float)
      position: character's global position
            x(float)
            y(float)
            z(float)
      range(float): range to character from my position
      velocity: character's global velocity
            x(float)
            y(float)
            z(float)
```

Game

```
mapname(string): name of terrain map
gamename(string): name of this game instance
gametype(string): name of the type of game
fraglimit(int): race to how many kills?
timelimit(int): time limit before game ends
maxclients(int): max number of characters
```

Map

```
node: list of all the pathing nodes in the environment
      position: global position of each node
            x(float)
            y(float)
            z(float)
      name(string): name of node
      connect-heading(float): direction to next node
left: readings for "feeler" sensor to the left
      range(float): distance until feeler hits something
      object-name(string): name of what it hits
      object-state(static/open/closed/moving): state of
what was hit (door open or close; static item; moving item)
right: readings for "feeler" sensor to the right
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
left-front: readings for "feeler" sensor to the left-front
(45 degrees to the left)
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
right-front: readings for "feeler" sensor to the right-front
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
front: readings for "feeler" sensor straight ahead
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
back: readings for "feeler" sensor straight behind
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
up: readings for "feeler" sensor straight up
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
down: readings for "feeler" sensor straight down
      range(float)
      object-name(string)
      object-state(static/open/closed/moving)
```

Sound

```
name(string): name of the sound
position: global position where the sound occured
      x(float)
      y(float)
      z(float)
volume(float): perceived sound volume
time-started(float): when was the sound first heard
```

Message

```
sender(string): who sent this message?
```

```
type(string): type of message
time-started(float): when was the message sent
phrase: content of the message
      word(string): first word of message
      next: link to next work
      word(string): next word of message
      next: link to next word...
            ...
```

In addition to the sensor values the HBM IS also specifies a set of actions outputs that the HBM uses to control a character in the environment. Each of these actions includes a number of parameters that inform the simulation as to the details of how to execute that action. For example, the move-to command includes an x,y,z location to be moved to and a speed with which to move. The action outputs and parameters are listed here with a brief description of each:

Action Outputs

```
move-to: move to the global x,y,z position at the speed
specified
      x(float)
      y(float)
      z(float)
      speed(float)
turn-to: turn the amount specified in the direction given
      direction(right/left/up/down)
      amount(float)      //degrees
centerview: bring the character's head and body in line
face: face towards the global x,y,z position
      x(float)
      y(float)
      z(float)
face-abs: face the global angle specified (0=north)
      angle(float) //degrees
attack: start attacking with the specified tactics
      value(normal/alt/charge/charge-alt/off)
            normal - start shooting
            alt - start shooting with the alternate weapon
                                   function
            charge - start shooting and run towards the
                                   target
            charge-alt - start shooting with the alternate
                                   function and run towards the
                                   target
            off - stop shooting
throw: throw the currently selected object with the strength
specified
       strength(float)
play-animation: play the specified animation
      animation(string)
play-sound: play the specified sound
      sound(string)
```

```
kneel: change posture to a kneeling position
lie-prone: lie in a prone position
stand: stand up from prone or crouching position
jump: jump once
brake: quit moving
side-step: move sideways
      direction(left/right/none)
thrust: move backward or forward
      direction(front/back/none)
move-target: continually move toward a specific enemy
      switch(on/off): start/stop moving towards the target
      target-name(string)
choose-weapon: select the specified weapon
      value(string): name of weapon
face-target: face the specified target
      switch(on/off): start stop facing the target
      target-name(string)
lead-target: face the point ahead of the target in the
direction the target is moving
      switch(on/off): start/stop facing target
      target-name(on/off): name of target
speak-to: send a message to a specific character
      phrase(string): message to send
      target(string): who is the target
      volume(float): how loud should I say the message
speak: send a message to anyone in the local environment
      phrase(string): message to send
      volume(float): how loud
toss-weapon: throw the currently selected weapon
reload: reload the current weapon
unjam: unjam the current weapon
surrender: play the surrender animation and stop fighting
```

4.0 Simulation Environment and Human Behavior Models

This section describes the Human Behavior Models and the Simulation Environment used in this effort. As the entertainment industry lab the effort described here was tasked with exploring HBM and simulation interfaces standards currently being used by the commercial game industry. To fulfill this task a commercial off-the-shelf (COTS) game system was chosen as the simulation environment. Unreal Tournament, a product of Epic Software, is a popular "first-person shooter" (FPS) game that includes one of the most widely used EI interfaces to allow hobbyists to extend and adapt (or "mod") the game. While Unreal Tournament, as purchased, is not a realistic simulation of urban combat, one such mod, Infiltration, changes the game to include more realistic weapon models, behaviors, and tactics. Section 4.1 describes Unreal Tournament in more detail and Appendix A describes the Infiltration mod.

In addition to the simulation environment, three human behavior models were selected. Representing the academic research community, the Soar system and Performance Moderator Functions were selected. Section 4.2 describes the Soar architecture and Section 4.3 describes Performance Moderator Functions. Representing the commercial game industry is AI.Implant, a game-industry middleware tool developed by BioGraphic Technologies. AI.Implant is described in more detail in Section 4.4.

4.1 Simulation Environment: Unreal Tournament

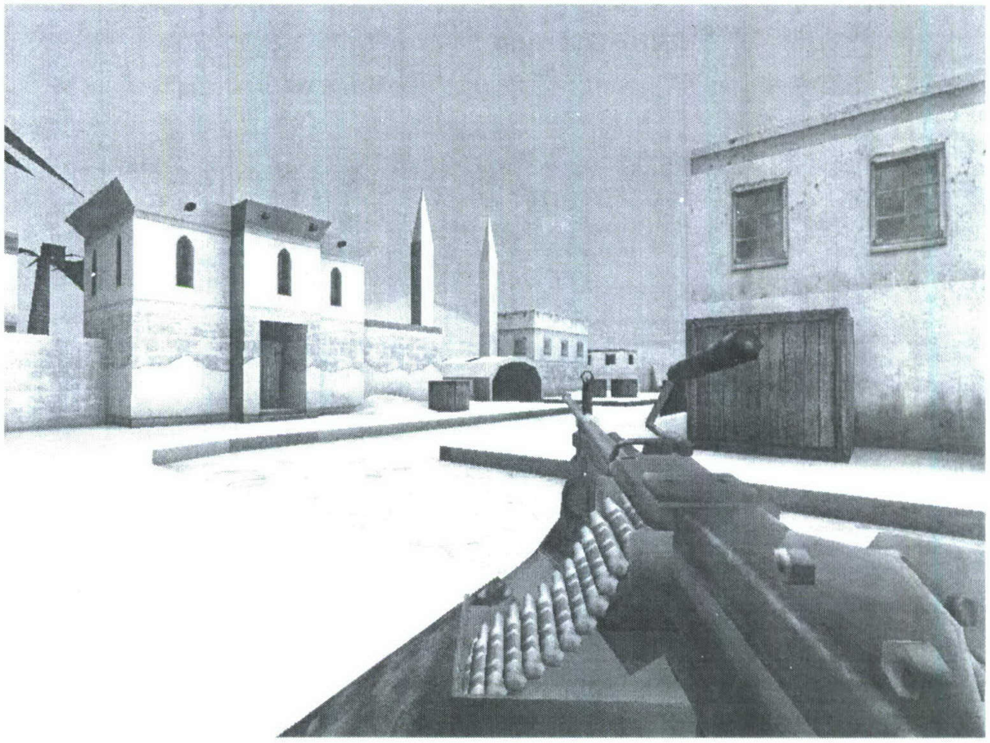
Unreal Tournament (UT) is the simulation environment targeted for the integration of all three HBM components. UT, released in 1999, was several industry publications' picks for Game of the Year. A first person shooter (FPS) game, UT includes a wide array of levels, weapons, and modes of play, and characters that are all modifiable.

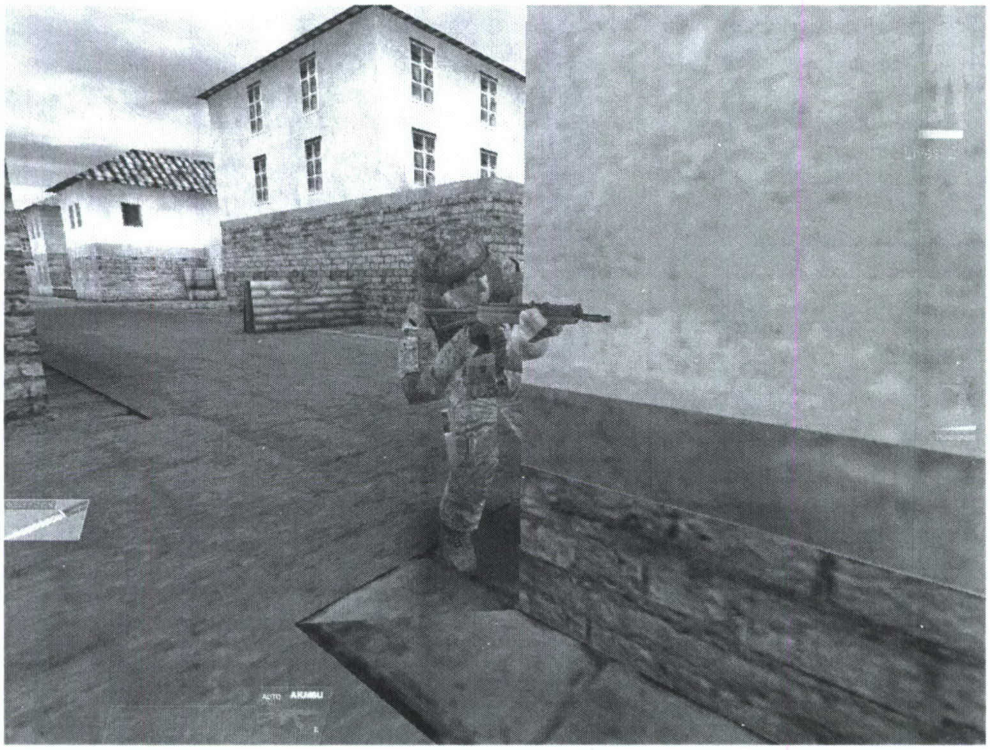
From a user's perspective, UT is extremely simple to operate. It can be as simple as clicking an icon to launch the game, selecting the map location, number of bots and their skill set, and clicking play. UT is capable of running on the PC, Linux, Macintosh, PlayStation2, and DreamCast. Other features that made it a viable choice for this particular project include:

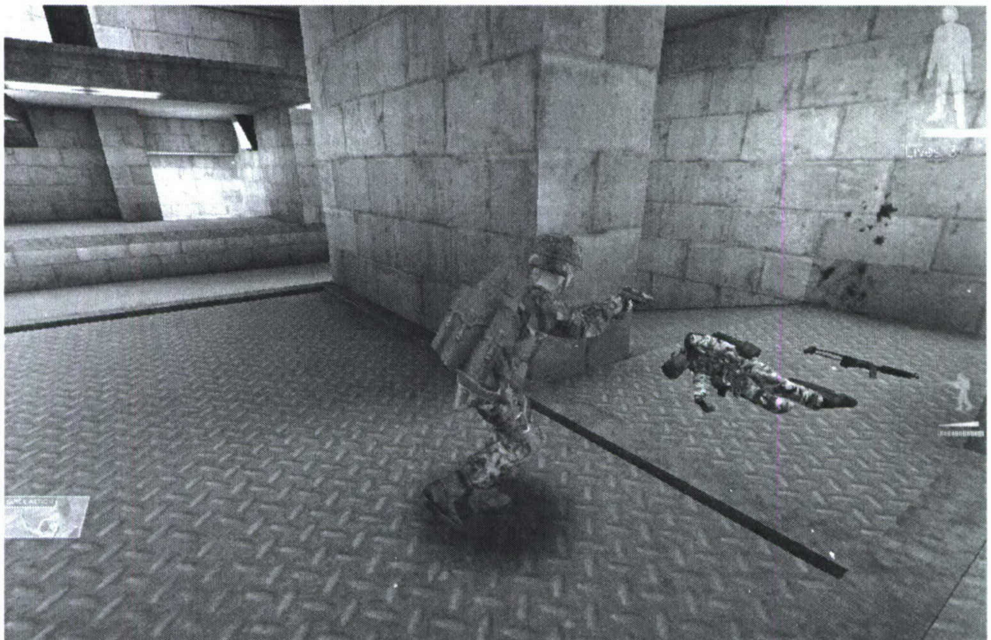
- Enhanced character AI — UT includes a built-in AI capability to control "bots" that play as opponents to the human user. Although the UT bots weren't used in the final demonstration they can be thought of as a fourth HBM operating in the simulation environment. These bots use a system of waypoints to navigate a level and control the character's AI. These waypoints are sub-classed to support a variety of tactical decisions and maneuvers (i.e. Defense, Ambush)
- Spectator Cams — these cams allow a user to enter a live game and navigate around to view the action on the level. This is an ideal way for soldiers to view how particular sequences within a scenario should be executed as well as replay the mission once it has ended.
- Publicly available "mod" interface — UT comes with a built-in software interface designed to support extensions and modifications of the game by the user

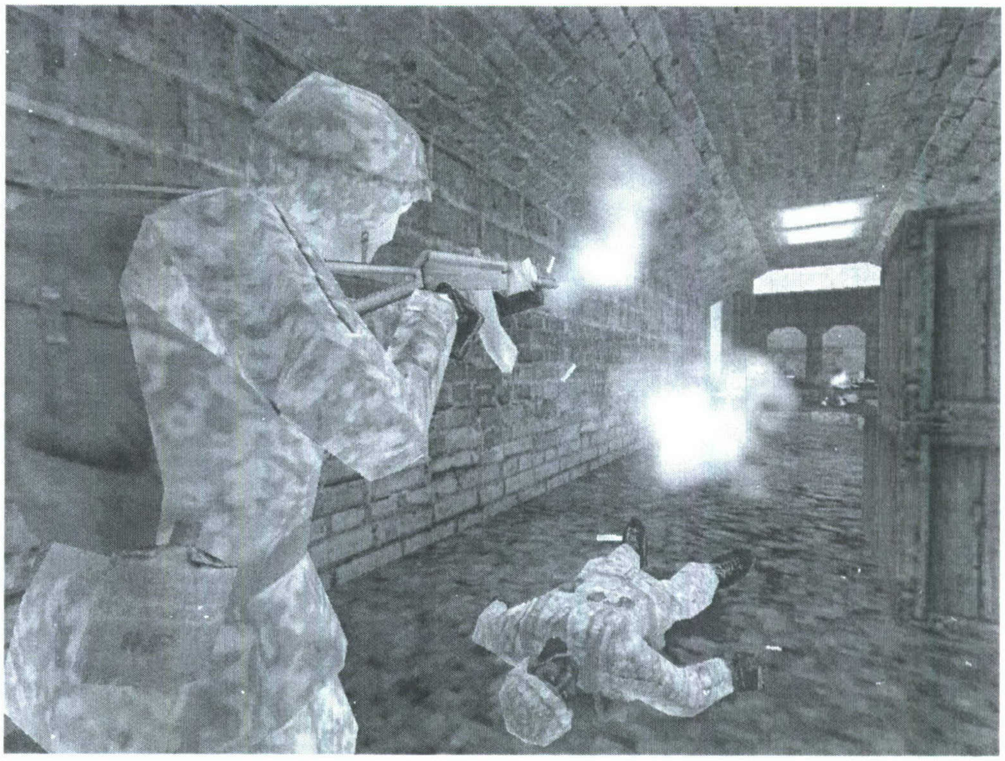
community. This "mod" interface represents one of the most successful game industry interface standard and serves as a starting point for the HBM IS.

The commercial Unreal Tournament release is being used for the HBM integration process. Using the standard release with the publicly available "mod" interface allows us to avoid purchasing a game engine license (approximate cost $350,000) from Epic software. As long as the software developed is not released as a commercial product no license fee needs to be paid to Epic. The commercial release includes the default weapon models, character meshes, and levels (terrain). However, because the release consists primarily of unrealistic weapons (laser guns, rail guns…), character models, and movements, a free "mod" to the game has been used to increase the realism of the game. This mod, *Infiltration*, replaces the futuristic weapons in UT with current U.S. Service-issued weapon types and ammunition. Moreover, the character models have been modified such that they display human skins rather than futuristic "bots." The "mod" feature of UT is described in more detail in the next section. Below are a set of screenshots from *Infiltration* that illustrate various aspects of game play. For more information on *Infiltration* and its capabilities, please refer to Appendix A.




AUTO AKMSU


FN FIVESEVEN



Unreal Tournament Game Engine (UTGE)

The Unreal Tournament Game Engine is the driver behind the UT simulation. Many of the UTGE components have been opened to the "mod" interface to give users a consistent programming interface, which allows access to many levels of the Engine (rendering, physics, AI, networking). UT was selected as the simulation environment due to this extensible and flexible interface allowing for fairly ease integration with external software modules. To interface an external software module (such as an HBM), an interface to allow the exchange of data is developed in UnrealScript (UScript). UScript is the high level language used to program game "mods", either from within the confines of the Engine or through an interface to an external program. UScript is an object-oriented scripting language that is very similar in syntax to Java. Many First-Person Shooter games have "mod" interfaces with custom programming languages (for example Quake has QuakeC). Unreal Tournament is unique in that it allows developers to extend UScript with native C++ code. This allows the research community to interface their existing system with the game engine rather than having to re-implement their HBMs in UScript.

A central aspect of the Unreal Tournament Game Engine and its execution is the notion of the game cycle. This term refers to each iteration through the game's simulation and rendering loop. To manage time, the UTGE divides each second of game play into "ticks." A tick is the smallest unit of time in which all actors/objects in a level are updated. Each tick usually takes between 1/100th and 1/10th of a second to execute and is limited only by CPU power (the faster the machine, the lower the tick duration is). It does *not* necessarily reflect frame rate, as rendering may sometimes be decoupled from the game processor. The UTGE cycles itself each iteration through the Tick() function, which is present in every Unreal class, either explicitly or inherited. Every part of the simulation that needs to be included in the update cycle is included through a call to the Tick() function for that aspect of the simulation. It is important to note that Tick() is called on *every* actor in the game each frame. This concept is central to the following sections of this report, as it manages how data flows and is exchanged between different aspects of the Engine, including HBM components.

Additionally, each time the game begins, there are a series of sequential steps that are followed by the game engine to initialize a variety of information about the simulation. Because this project involves a significant amount of character creation, Actor scripts were created for each HBM character-type, which are standalone object modules that determine all aspects of a character's appearance, weapon type, AI, movement, tactics, and relationship to other users. Below is a sequential list that describes user initialization, from the time a character is spawned through each Tick() of the game cycle:

1. The actor object is created (character is placed in the level) and its variables are initialized to their default values (if spawning an actor), or loaded from a file (if loading a pre-existing actor)—these variables include weapon type, initial position, ammunition, maximum speed, health, etc.
2. If the actor is being spawned, its Spawn() event is called—this creates an instantiation of a particular class that may be manipulated the same way an object is; allows multiple classes to be created and called within a single class
3. The actor's PreBeginPlay() event is called—this function, inherited from the Actor class, is called by the engine on every Actor immediately before the game begins. One action usually taken within this function includes setting up and initializing class variables
4. The actor's PostBeginPlay() event is called – also inherited from the Actor class, this function is called after PreBeginPlay() and immediately after game play begins. It too is called by the engine and used to set up variables, especially those that require instantiation after the game has already been initialized
5. The actor's SetInitialState() event is called—called immediately following PostBeginPlay(), this directs the engine to set the initial *state* of the character, such as walking, running, attacking, or waiting
6. Tick() is called each iteration through the game cycle—developer-defined functions may be called here to manage the character's behaviour, movement, or AI.

This information is important to keep in mind when reading through the remainder of this document as it is used to describe how the various HBMs operate in the context of the game engine. Though very few of the functions are explicitly cited, it helps to understand the iterative process that the UT executable follows from initialization throughout the play of the game.

Pros/Cons of the Unreal Tournament Game Engine

There were several advantages with utilizing the Unreal Tournament Game Engine, such as its existing interface to Soar. Below is a list of additional advantages for using Unreal Tournament as opposed to other game-based environments (i.e. Quake, Operation Flashpoint, and Battlefield 1942). A comparison of Service-based simulations and architectures and their applicability to this project is also presented below. The criteria for evaluation are primarily technical and centered on the scalability and flexibility of incorporating disparate HBM applications and their data into a commercial simulation environment.

- A well-developed, industry-tested C++/UnrealScript interface, enabling development of complete projects in either language
- Highly modularized and replaceable, with all Unreal Tournament game code cleanly segregated from all general Engine code—this allows developers to create modules that are clearly separate and distinct from the Engine itself
- A C++ interface based on an object model that is similar in style to Microsoft Foundation Classes (MFC), which is the application framework used by nearly all Windows-based applications—provides many developers with a rapid development environment to create UT plug-ins
- Supports dynamic loading of Dynamic Link Libraries (DLLs) and scripts on demand, for modularity and efficient memory storage
- Robust debugging environment, with Visual C++ debugger support; flexible assertion system; a try-catch call stack display for tracking down errors in the field—all of these benefits decrease testing time during the development phase
- An UnrealScript interface based on an object model that is similar to Java (inheritance, polymorphism, garbage collection)—a well-known industry standard that is familiar to many developers

In addition to these advantages, there are a couple of disadvantages with the Unreal Tournament Game Engine. For example, when using native function extensions to UScript, any external applications that you integrate into the engine will be tightly coupled to the game cycle. This makes incorporation into any other simulation environment much more difficult as two competing simulation cycles will need to be integrated and deconflicted.

4.1.1 Modding and Native Functions

To integrate the various HBMs with the Unreal Tournament environment changes had to be made to the game executable (*not* the Engine) which allows these pieces to interact with the game environment. These changes are made by making modifications (or mods) to the game through UScript, described above. One example of an advanced mod is "Infiltration," which was the game-type selected for this project because of its added realism and closer resemblance to actual military combat.

Infiltration is a squad-based, mission-oriented add-on that utilizes modern-day weapons (M16-A2), equipment (Kevlar Helmets) and tactics (strafing around corners).

There is no crosshair floating in front of the character that is typically seen with First Person Shooter (FPS) games; instead, the weapon must be drawn to the user's eye and aimed through the iron sights or scope. Finally, movement has been changed to restrict users from advancing any faster than typical humans would on the battlefield. Users have the ability to crouch, get in a prone position, and lean around corners. One of the biggest complaints typically heard from using games such as this within a military domain is the lack of realism. Though this modification is still an entertainment-oriented environment, it contains a good deal of realism and has the potential to support a wide variety of research efforts. As these research efforts mature they can then be transitioned to military simulation systems and undergo VV&A.

Using the software development principles of inheritance, polymorphism and "garbage collection," developers have the option of modifying existing UScript classes, adding their own classes, or interfacing to external software modules. The latter occurs through the definition of native functions within UScript, which are calls to external C++ classes defined by the developer. In order to call these native C++ functions from UScript, you must compile them into a DLL file. It is important to note that the process of implementing native functions is not supported by Epic Software and Epic has not documented the process. However, Epic has encouraged the user community to provide a wide range of documentation and tutorials that are available on the web. The header files that are generated by the UT compiler (ucc) are strictly in C++, eliminating the possibility of using another language—unless wrappers are explicitly created for that purpose. The PMFServ interface is an example of using software wrappers to integrate another language, in this case Python.

4.1.2 Unreal Tournament vs. DoD Simulations

Based on our experiences integrating three HBM components (Soar, AII, PMFServ) into the Unreal Tournament Game Engine, coupled with previous experience with military simulations, it is possible to make some general, qualitatively comparisons between the two simulation environment.

OneSAF Testbed Baseline (OTB)

The OTB Semi-Automated Forces (SAF) System is the successor of ModSAF. It supports a full range of operations, systems, and control processes from the individual combatant and platform up through brigade-level operations. The various components that make up OTB SAF communicate physical battlefield state and events among themselves through the simulation Distributed Interactive Simulation (DIS) protocol. Written primarily in C (with separate Java plug-ins), this platform level simulation environment has some similarities and many differences to UT.
Similarities

- Both simulations support entities that are capable of acting totally autonomously
- Both can execute a realistic set of basic actions inherit to the type of entity simulated. For example, in OTB SAF, a tank can follow a path along a road; in UT, a character can navigate through a series of rooms

- Both [can] exhibit realistic weapon rates of fire and trajectories, and resource depletion is accurately simulated for ammunition
- Both contain target detection, target identification, target selection, and collision avoidance and detection capabilities
- Both can receive a set of orders/commands and generate the appropriate entity/character behavior and tactics without further user action

Differences

- Whereas UT runs on a variety of platforms, including Windows, Macintosh and Linux, OTB SAF is limited to certain Unix distributions, Linux (Debian $\geq$ v2.1 or RedHat $\geq$ v6.0), and Windows NT/2000 (only with the appropriate UNIX emulation software). This is probably the most important difference between the two as the HBM IS was developed in a Visual C++ environment using Win32-based APIs. Moreover, the actual software is a DLL that runs only on Windows machines
- UT does not make use of verified, validated, and accredited (VV&A) models for damage assessment, delivery accuracy, and target acquisition. OTB SAF models are compatible with existing validated Army combat models
- OTB SAF: User has control over every entity playing in the simulation; UT: User only has control over himself (and possibly subordinate units if included)
- OTB SAF: A "plan view display" of the simulation—the ability to navigate around the entire area and manipulate entities; UT: Restricted to a first-person perspective.
- OTB SAF: User can override any commands issued to an entity/unit by the SAF AI; UT: User only has direct control of his character.
- OTB SAF: Realistic terrain, weapons, and physics models; UT: Most models were designed for entertainment and not for realistic training scenarios, so many of them are limited in realism.

Although the differences cited above are significant Unreal Tournament, and other game engines, should not be dismissed as useless other than for entertainment. Games are very useful as virtual environments for early research efforts that are not yet ready to undertake the significant effort required to be integrated with something like OTB. In addition to the significant software challenge inherent in this integration there is also a great deal of knowledge acquisition required to encode the necessary behaviors and, finally, an extensive VV&A process. Games offer a cheap, fairly complex, flexible and attractive environment in which to explore initial ideas while still staying in the general realm of military applications.

Dismounted Infantry SAF (DISAF) is a variant of ModSAF that focuses on simulation at the individual soldier level. Because DISAF is the DoD simulation architecture that most resembles Unreal Tournament we've undertaken an operational comparison of the two systems. DISAF was developed to add dismounted infantry to the virtual battlefield and be compatible with other Human-In-The-Loop simulators. Below is a table that illustrates the areas that DISAF currently addresses in comparison to UT.

		DISAF	UT
Forces:	Friendly, Enemy, Neutral	All	All
Unit Echelons:	IC, Fireteam, Squad, Platoon	All	None
Postures:	Prone, Sitting, Kneeling, Standing	All	Prone, Kneeling, Standing
Weapon States:	Stowed, Deployed, Raised	All	All
Sensors:	Visual (eyes) and Aural (ears)	All	All (though simplified)
Weapons:	AT8 Missile, SAW, M203 Grenade Launcher, M16A2, Hand Grenades, AK47	All	All but the AT8 Missile (which could easily be added)
Terrain:	Multi-story Buildings with Interiors Apertures (Windows, Doors, and Blown Holes in Wall), Dynamic Terrain (Building/Aperture hits)	All	All

High Level Architecture (HLA)

Though this particular effort is highly research-centric, it is important to address the issue of HLA and its relationship to the HBM IS. HLA is a general purpose architecture developed to support the reuse and interoperability across large numbers of different types of simulations developed and maintained by the DoD. OTB version 1.0 is one example of an HLA-compliant DoD simulation. One important aspect of HLA is the notion of a federation; a federation is a set of simulations intended to "play together" to form a larger model or simulation. Each simulation, support utility, or interface is a federate that is a subset of the federation. It could represent one platform, such as a cockpit simulator, or represent an aggregate of forms, such as an entire OTB exercise executing on numerous individual platforms.

HLA consists of three primary components: HLA Rules, Interface Specification, and Object Model Template. The Rules must be followed to achieve proper interaction of simulations within a federation. They describe the responsibilities of simulations and of the runtime infrastructure in HLA federations. The Interface Specification is the definition of the interface functions between the Run Time Infrastructure (RTI) and the simulations subject to the HLA. Finally, the Object Model Template is the common method for recording the information contained in the required HLA Object Model for each federation and simulation. There are a series of documents that describe each of these definitions in more detail.

As reflected in this brief description, HLA provides a methodology for designing and developing DoD simulations. With regards to UT, which was designed far outside the realm of a DoD mandate, there is little in common with HLA. However, with the availability of the Unreal Tournament Game Engine source code (in C++), reworking various aspects of the engine to make it HLA-compliant would not be impossible (but certainly not trivial). Moreover, the HBM components would probably act more as Support Utilities existing as separate software modules rather than being incorporated directly into the simulation, as is currently the case. This would support the inclusion of the HBMs into other HLA-compliant simulations, though significant work would be required to determine the correct domain and scope (i.e. platform vs. unit level simulation).

4.2 HBM: Soar

As described in the Soar tutorial, Soar is "a unified architecture for developing intelligent systems. That is, Soar provides the fixed computational structures in which knowledge can be encoded and used to produce action in pursuit of goals. In many ways, it is like a programming language, albeit a specialized one. It differs from other programming languages in that it has embedded in it a specific theory of the appropriate primitives underlying symbolic reasoning, learning, planning, and other capabilities that we hypothesize are necessary for intelligent behavior. Soar is not an attempt to create a general purpose programming language. You will undoubtedly discover that some computations are more appropriately encoded in a programming language such as C, C++, or Java. Our hypothesis is that Soar is appropriate for building autonomous entities that use large bodies of knowledge to generate action in pursuit of goals."

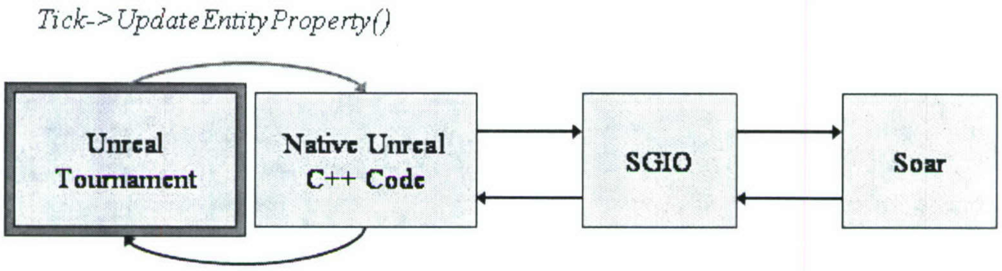
The integration of Soar into UT for this project was modeled after an existing Soar-UT integration (MOUTBot). This section will describe the relationship between the two components and how they exchange information. There are several classes of information that are retrieved from the environment as sensor input to the Soar architecture, namely:

- Information about the Soar-controlled character (agent)
- Feedback on the actions sent to the environment (feedback)
- Information about dynamic objects in the environment (objects)
- Information about the game parameters (game)
- Spatial representation of the static terrain (map)
- Information on any sounds in the environment (sound)
- Communication between the characters (HBM or human controlled) (message)

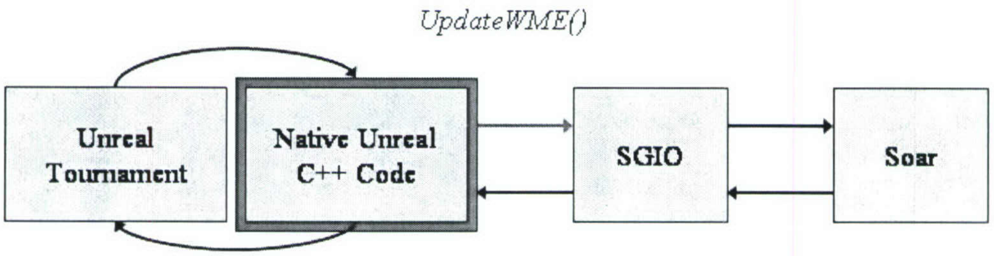
Upon initialization of a Soar-controlled character, this information is retrieved and stored in Soar's local memory on the "input-link". This input link is built based on what the SoarBot can "sense" through a realistic set of sensors. The SoarBot can't see through walls or sense aspects of other characters that aren't available to a human user.

Each tick through the game cycle, the Unreal Tournament Game Engine (UTGE) passes to Soar any updates from the environment. If any elements of the environment are different than those stored in local memory, they are updated on the input-link and the cycle repeats itself. The actual implementation of this process occurs through the C++ library developed by Brad Jones (University of Michigan) called Soar General Input/Output (SGIO). SGIO allows various environments (including UT) to communicate with Soar through a common DLL or socket interface. The design of SGIO

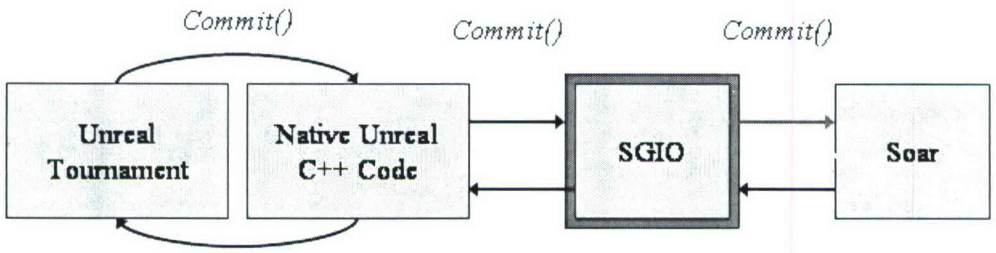
is centered on three main classes: Soar, Agent, and WorkingMemory. The Soar class represents a connection to Soar. The Agent class represents a particular agent being controlled by an instance of Soar. Finally, the WorkingMemory class is a utility class designed to aid users in managing some of the "bookkeeping" associated with an Agent's WorkingMemory.
Below is a sample run-through of a SoarBot Game Cycle. The process is procedural and may be followed by the descriptions at the bottom of each illustration.



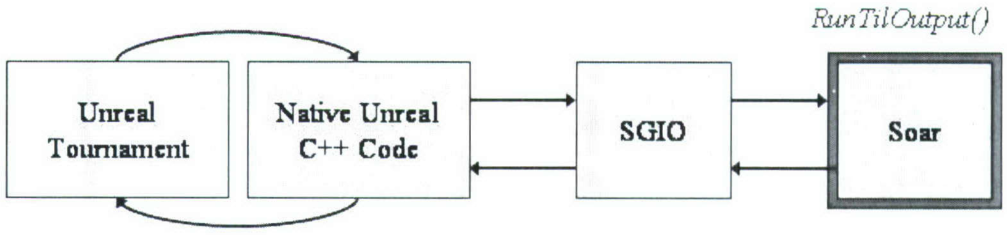
The Tick() function is called each iteration through the Game Cycle. If any entities within the environment have changed, a call is made to the native C++ function, UpdateEntityProperty().



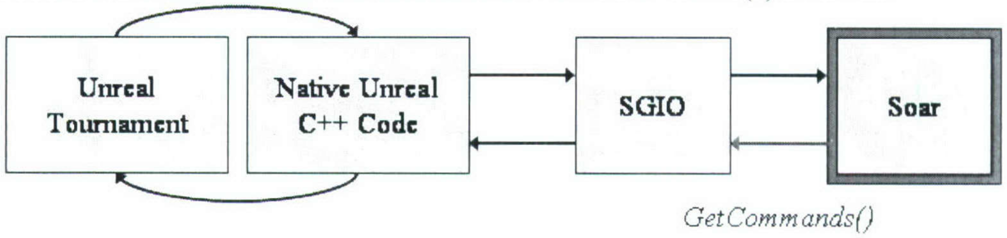
The updated entity information is now in the native function and must update Soar's input link. The native DLL sends this updated information to SGIO through the UpdateWME() function.



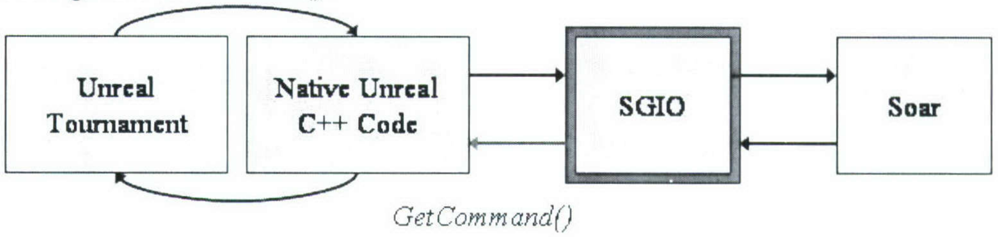
Once all the sensor updates have been collected from the environment and the Working Memory Elements (WMEs) have all been updated, the WME changes may be committed to Soar. This is the first point in the game cycle where environment information has explicitly been loaded into the Soar Architecture.



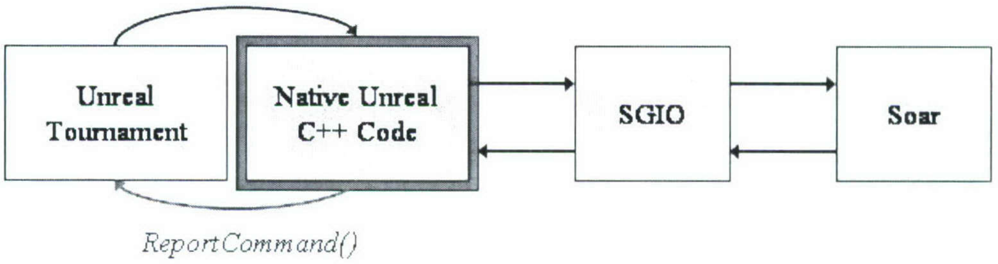
Soar processes the information and determines which command(s) to issue.



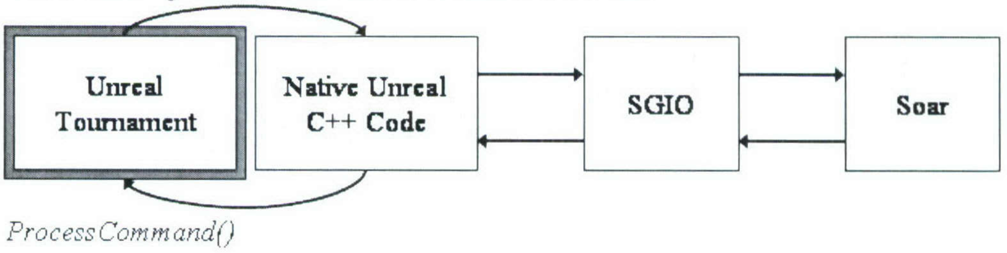
After this determination has been made, the appropriate command(s) are retrieved by SGIO through GetCommands().



The command at the top of the queue is passed back to the native Unreal code each Tick().



The native DLL reports the current command to Unreal.

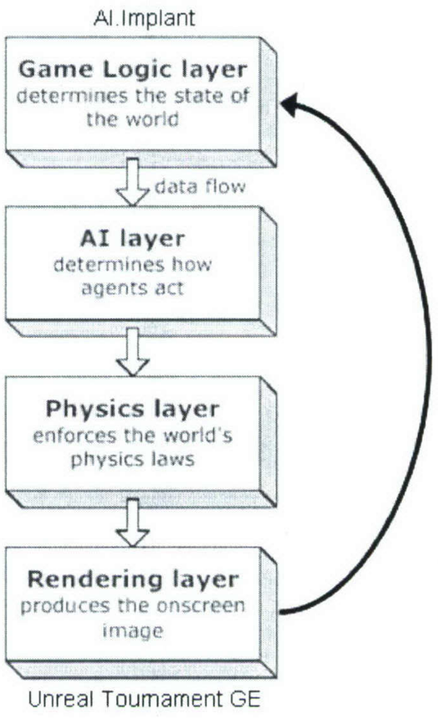


The UTGE processes the commands (i.e. move, fire) resulting in the Soar-controlled character performing actions in the environment. The cycle then repeats itself each iteration through the game cycle.

The development of the SoarBot interface and behaviors was undertaken both by ICT and the University of Michigan. The pre-existing MOUTBot was used as a baseline, though significant changes were made that allowed the Bot to operate as a subordinate NPC to the human user. A list of all the changes made to the MOUTBot can be found in Appendix B.

4.3 HBM: AI.Implant (Proof)

AI.Implant (www.aiimplant.com) is a 3rd party commercial toolset that allows developers to build and control in-game characters, specifically group behaviors and basic navigation and path planning. It simplifies the need for the programmer to manually define particular behaviors and movements at the lowest level, allowing for enhanced game play and more intelligent characters. AII allows for the creation of complex action sequences for AI agents, which interact with the game engine in the following manner:


AI.Implant
Game Logic layer
determines the state of the world
data flow
AI layer
determines how agents act
Physics layer
enforces the world's physics laws
Rendering layer
produces the onscreen image
Unreal Tournament GE

This data flow diagram illustrates how the AI layer (represented as AII objects) interacts with the various layers of the game engine (GE). The Game Logic Layer receives information on the state of the game and passes this to the AI Layer, both of which are represented within the AII environment. After receiving information on the "state of the world" (essentially the location of objects, both static and dynamic), the AI layer is capable of determining how the agents are going to behave and move. This occurs through the Software Development Kit's (SDK) Solve() function, which is called each iteration through the game cycle (every time Tick() is called). After movement and behavior information have been computed, the information is passed to the Physics layer (part of the Game Engine), which determines exactly how the AI will be executed within the Engine. From here, the information is rendered and the cycle repeats itself.

The AII distribution comes with the SDK and associated documentation. The SDK consists of a suite of C++ Application Programmer's Interfaces (APIs) that developers integrate with to create game AI within their software. These APIs create complex action sequences for AI agents (characters) within the simulation environment. This occurs by establishing basic characteristics (general constraints) of the agent, adding defined behaviors (steering forces), and enabling the agent to make decisions via sensors and decision trees. The best way to look at AII is as middleware tool that is linked with a game engine to improve AI performance within a game. Because it is a relatively new product, the documentation that is released with the SDK is not too descriptive, making integration into any Game Engine rather difficult. However, many of the functions are self-explanatory and provide developers with a rich suite of tools to control AI within a simulation.

AII uses a fairly novel way of representing the world terrain. Where classic game industry methods involve creating a path node or waypoint network for an agent to follow, AII adopts a higher level approach. Barriers, walls that an agent can see and react to, are used to create an approximation of the actual walls and objects in the UT level. So, instead of an agent being forced to follow specific paths between nodes they can now use the walls as indicators of where they can and cannot go. These barriers can also be used to automatically generate a path node network if one is needed.

The AII framework provides many of the base behaviors and functionality needed to achieve the functionality needed in most games. Using the SDK also gives you the opportunity to define new behaviors or extend existing behaviors. Given enough time the functionality could be extended to nearly anything that was required. Using the SDK directly also gives you the option of overriding core components (such as wall detection), essentially tightening the link between AII and the existing game engine. However it is unknown as to how much extra load this would place on the game engine and it may prove inefficient.

There are several drawbacks to working directly within the AII SDK. The largest drawback is that backwards compatibility is not guaranteed for SDK code when the AI.Implant system is updated. Thus, when BioGraphics releases a new version extensive effort may be required to get previously working systems back in working order. This had a significant impact on the development schedule for the HBM IS project. Secondly, there is actually little to no documentation beyond header files. However, the developers at BioGraphics provide fairly good support through email and by phone. Lastly, working within the SDK requires advanced knowledge of AII's data structures and their interaction. This extra knowledge is required just to get things functioning inside AII and holds no bearing on the overall design of the interface, characters, behaviors, and decision trees. Essentially the SDK is provided as a means to extend the functionality of the system, it is not meant to be used to code the actual world.

Due to the nature of the AII SDK it would be a wise to consider using the Maya/3Ds Max plugins to generate an ACX file. The ACX file is an AI.Implant specific file format that encodes AII behaviors. This ACX file can be imported into the SDK and then the relevant pieces can be stitched together. This approach should support backwards compatibility with future versions of AI.Implant. The only drawback to this approach is the requirement of Maya or 3Ds Max software which can be expensive (approximately $800/user for 3Ds Max). Biographic Technologies is currently working on a standalone editor application to fill this gap which, upon release, would make working directly with the SDK more viable.

The following is a list of the AI.Implant behaviors that are available "out of the box." Additional behaviors can be developed either by combining and building off these canned behaviors or starting from scratch.

Basic navigation:
- ACE_BehaviourAvoidBarriers—prevents the character from colliding into barriers (walls, buildings)
- ACE_BehaviourAvoidObstacles—prevents the character from colliding with game obstacles (vehicles, doors, trees)
- ACE_BehaviourAccelerateAt
- ACE_BehaviourMaintainSpeedAt
- ACE_BehaviourWanderAround—a random wander behavior that allows a character to aimlessly meander around a level
- ACE_BehaviourOrientTo

Group behaviours:
- ACE_BehaviourAlignWith
- ACE_BehaviourJoinWith
- ACE_BehaviourSeparateFrom
- ACE_BehaviourFlockWith

Targeted behaviours:
- ACE_BehaviourSeekTo
- ACE_BehaviourFleeFrom
- ACE_BehaviourLookAt
- ACE_BehaviourFollowPath
- ACE_BehaviourSeekToViaNetwork

State change behaviours:
- ACE_BehaviourStateChangeOnProximity
- ACE_BehaviourTargetStateChangeOnProximity

5.0 Final Demonstration

Software: The demonstration is provided in both AVI and Quicktime Movie formats. Quicktime works reliably on both Windows and MacOS machines but is a less compact format. The AVI format works well on Windows and is a more compact format but is less reliable on MacOS. A copy of the free Quicktime Player for Windows is included with the demo. MacOS machines should come pre-configured with the Quicktime

Player. To install Quicktime on a Windows computer simply unzip the files contained in QuicktimeInstaller.zip and double click on the executable "QuicktimeInstaller.exe."

Demonstration Description: The DMSO Human Behavior Model Interchange Standard project demonstration starts with a series of video captures highlighting each aspect of the HBM IS project including the demonstration environment, the AI.Implant, Soar, and PMFServ human behavior models. The demonstration ends with a video capture of the tactical encounter that forms the heart of the HBM IS demonstration. The full demonstration runs approximately 6 minutes.

The HBM IS demonstration uses the Unreal Tournament game engine created by Epic Software. A freely available modification (or "mod") to the Unreal Tournament game, called Infiltration, is used to increase the realism of many aspects of the game. These include weapon models, character models and behaviors, game dynamics (such as injury models and realistic sight picture for aiming). The Human Behavior Model Interchange Standard works through Unreal Tournament's publicly available "mod" interface and didn't require a game engine license to be purchased from Epic.

Mogadishu Level Fly-by: The first sequence presents a fly-by overview of the Mogadishu-themed urban terrain that was custom built by Quicksilver software for the DMSO HBM IS project. This is approximately 16 city blocks that have the "look and feel" of Mogadishu and contain specific terrain features from the movie BlackHawk Down including a downed BlackHawk helicopter. As the level currently stands users and HBM agents are not able to enter building interiors.

AI.Implant: The next sequence of the demo presents the behavior of the AI.Implant control opponents. AI.Implant is a game industry middleware tool that focuses on navigation and path planning in 3D environments. Whereas most games use an invisible waypoint graph spatial representation, AI.Implant uses a barrier-based spatial representation. By extracting barriers directly from the 3D environment, AI.Implant can fine tune its movement within that environment and doesn't require a level designer to reconfigure a waypoint graph each time the level layout changes. In the demonstration sequence the user (in invisible, free-fly mode) follows the AI.Implant-controlled militia leader. The leader travels in a circular patrol path in the Mogadishu level. As the AI.Implant-controlled militia members spot the leader they move towards him and follow him through the level using AI.Implant's built in flocking behavior.

Soar (University of Michigan): The next sequence presents the first set of Soar-based behaviors used to control the four soldiers of the fire team that team with the human user. The University of Michigan Soar behaviors, shown in this sequence, are encoded entirely in Soar productions. The ICT Soar behaviors, shown in the next sequence, are encoded in a combination of high-level Soar productions and mission-specific extensions to the HBM IS. In the demonstration the human user controls the center character while the four characters in the surround box formation are Soar controlled. As the user moves the Soar-controlled agents follow and seek to re-establish the box formation. The human user then orders the Soar-controlled agents into a line formation and demonstrates how the Soar agents follow and re-establish the line formation. The human user then orders one pair of Soar agents to hold position while the second pair of Soar agents continues to move with the user. The human user then orders all of the Soar agents to hold position and moves with the Soar-controlled soldiers following. Finally, the Soar agents are again ordered into the box formation and move with the human user.

Soar (ICT): The next sequence demonstrates a second set of Soar-based behaviors developed at ICT. These behaviors demonstrate how the base HBM IS (used in the University of Michigan behaviors) can be extended to include higher-level, mission-specific behaviors. While productions are still used to control the high-level behavior of the Soar agents, lower level behaviors, such as maintaining a formation, are now implemented through mission-specific extensions to the Human Behavior Model Interchange Standard. While these extensions are specific to the formations used in this demonstration, they are not Soar-specific and could be used by other Human Behavior Models. Again the center character is controlled by the human while the four surrounding characters (in white camo in this case) are controlled by the ICT Soar behaviors. The demonstration starts with the Soar agents moving with the character in a box formation. Because the low-level formation holding behaviors are encoded in the HBM IS they are able to react to the user's movements much more quickly. This results in much tighter, but less natural (almost robotic), movement of the formation. The Soar agents are ordered into a line formation which displays similar characteristics. As before, individual pairs of Soar agents and all four Soar agents are ordered to hold position.

3 Simultaneous HBMs: Soar, AI.Implant, PMFs: The next sequence demonstrates all three Human Behavior Representation Models acting in the same environment, at the same time, through slight variations of the base HBM IS. To our knowledge, this is the first time three different HBMs have operated in a single environment at the same time. At the start of the sequence the human user is surrounded by his fire team of four Soar-controlled characters. The AI.Implant-controlled opponents are visible in the distance (and through the sniper scope). The PMF-controlled "gasping bot" is the character in white camo with no hat near the human's team. The user switches to the PMF control screen and lowers the energy reservoir to cause the PMF agent to start gasping. The user then lowers the energy again causing the PMF agent to collapse. The rest of the sequence continues to demonstrate all three HBMs operating at the same time. In addition to the "gasping bot", a full set of PMF behaviors were demonstrated separately by the University of Pennsylvania team.

Final Demonstration: The final sequence demonstrates a tactical encounter between the user's Soar-controlled fire team and the AI.Implant-controlled opponent militia. In this instance the ICT Soar behaviors are used. The user starts by waiting until the AI.Implant agents have turned the corner and then moves up behind them and tells one pair of Soar agents to hold position as a base of fire. The user then moves with the other pair of Soar agents into position to flank the AI.Implant militia. The user then waits for the AI.Implant agents to move through their next patrol loop (not that we've reduced AI.Implant's line-of-sight distance for demonstration purposes). The user and two Soar agents then move forward to catch the AI.Implant militia in a crossfire with the other two Soar agents.

6.0 <u>Future Work</u>

The work done in the context of the HBM Interchange Standards Project suggests a number of research areas that show potential and are likely to be fruitful topics for future work.

6.1 Games as Research Environments

For a number of years now commercial games have been used as research environments for a variety of research areas. Commercial games provide cheap, easily accessible virtual environments that are almost as complex as large scale simulation systems such as OneSAF and JointSAF. Many of the problems faced in these simulation systems can also be explored in the context of one or more commercial games. Games are also an attractive domain for graduate students and undergraduates and make for compelling demonstrations. More than 20 major research universities and institutions use games as environments for ongoing research efforts. These include the University of Michigan, Carnegie Mellon University, University of Southern California, Northwestern University, SAIC, BBN, and MAK. Many of these organizations are using the Unreal Tournament game engine due to the native function support mentioned above.

In most games realism is less important than entertainment. However, some realism mods, such as Infiltration, and simulation-based games, such as flight simulators and Sonalysts' naval simulations, rival even the most realistic simulation systems. For many research projects the realism of the environment might be less important than the easy of integration with that environment. For the early stages of research an inexpensive, easy to use, complex, compelling virtual environment, based on a commercial game, might be the most effective environment. ICT researchers will continue to use commercial game engines as research environments. We hope to build on our experiences with First-Person Shooter game engines to expand into other game genres such as Real-Time Strategy (RTS) games and Massively Multiplayer Online Role-Playing Games (MMORPG). Unfortunately these game genres currently lack the publicly available "mod" interfaces that are common in FPS games. ICT has discussed building HBM interfaces into existing games with a number of game companies, most notably Quicksilver, developers of Masters of Orion III, and UbiSoft, developer of the upcoming Matrix MMORPG. One key step in shifting to new game genres is identifying research areas that can be effectively explored in the context of that genre of game and then transitioned to other systems and used for training, analysis, mission rehearsal, or any of a number of other DoD purposes. ICT has been involved in a recent DMSO funded effort to identify analysis applications of MMORPGs headed by Dr. David Johnson at IDA. This effort has suggested that the involvement of large DoD organizations, such as DMSO, can be helpful in getting and keeping the attention of commercial game developers.

6.2 HBM Interchange Standard

The most significant result of the HBM Interchange Standards project is the demonstration that three very different HBM models, coming from two different communities, can operate in the same environment through a single interface. The prototype HBM Interchange Standard developed in this project has shown some potential and further development of this interface seems a valuable area of future research. ICT and UPenn will likely continue this work in an informal manner as a part of future HBM related projects. Of particular interest is extending the interface to cover different types of spatial reasoning (see next subsection), command echelons, game genres, and HBM models (such as ACT-R).

6.3 Spatial Representation

One interesting outcome of the comparison of AI.Implant, Soar and the NPC AI internal to Unreal Tournament was the discovery that these systems use three very different approaches to spatial representation. The Unreal Tournament NPCs use the common game industry approach of embedding invisible "pathing waypoints" in the map and connecting the waypoints with edges to create a waypoint graph. This waypoint graph is constructed such that an NPC can move throughout the map by moving from waypoint to waypoint along the edges. Two waypoints are connected by an edge if there is no blockage between the waypoints. Thus, by following edges the NPC will never run into an obstacle and doesn't even need to know that the obstacles exist. The waypoints and edges are generally added by hand usually by the designer of the map.

AI.Implant takes the very different approach of explicitly representing the obstacles in the world by surrounding them with barriers. Ideally, these barriers can be extracted directly from the polygons that make up the world through AI.Implant's interface with the Maya 3D modeling program. However, if the environment isn't modeled in Maya extracting barrier information can be very difficult. For the Unreal Tournament level a set of barriers had to be input by hand to surround all the obstacles in the world. Internally AI.Implant uses an automatic algorithm to generate a waypoint graph that avoids the barriers and fills the space. Thus the waypoint graph representation is again used but not created by hand.

The Soar architecture uses "corner nodes" and "door nodes" to represent the terrain. Each room has two corner nodes, in opposing corners, and a door node on either side of each door into the room. As with Unreal Tournament these nodes are placed by hand. This representation gives Soar exact information about the dimensions of each room and the location of each doorway allowing the Soarbot to move throughout the environment without being limited to a pre-defined waypoint graph. The Soarbot can also use this spatial representation to perform more sophisticated spatial behaviors such as dodging behind corners, popping out of doorways to shoot and then ducking back in, and circling around to attack from behind. A static, pre-defined waypoint network would make many of these behaviors difficult as the dynamics of the behavior are very dependent on the specific position of the bot and opponent.

Further exploration of the benefits of these different spatial representation approaches, as well as approaches used by other HBRs, is a good candidate for future work. It may also be valuable to consider creating a standard for spatial representations that, like the HBM Interchange Standard, could be shared between commercial game developers and DoD systems.

6.4 Combined Human Behavior Model

The development of a CHBM is useful in that it often identifies conflicts, gaps, or inconsistencies in the individual component models that comprise it. The CHBM developed for this effort focuses on goal-directed tactical behavior, physiological and emotional effects on behavior, and navigation.

The CHBM consists of components taken from both the academic research community and the computer game industry. The Soar architecture (University of Michigan) acts as the central behavior generation component. Performance Moderator Functions (PMFs), developed under the guidance of Dr. Barry Silverman at the University of Pennsylvania, will interact with the existing simulation's Artificial Intelligence (AI) to reflect physiological and emotional states. AI.Implant (AII) (BioGraphic Technologies) acts as the primary path-planning and navigation layer for characters within the environment. Each of these components are the most sophisticated in their specific domain but have been developed somewhat in isolation. The primary focus of the ICT team is the design and development of a number of interchange standards and specifications that will allow the three components to work together as a single architecture and to interact with the simulation environment. For this phase of the project, the primary simulation environment is a version of Unreal Tournament (UT) that has been modified to support more realistic military environments, equipment, and tactics (*Infiltration*), which is described below.

7.0 Deliverables

This section details the deliverables of the DMSO funded Human Behavior Representation Interchange Standards Laboratory and when each was completed. These deliverables are taken from section 1.6.3.4 of the "Vision Project 2002, Innovation Workshop, and Laboratory for Human Behavior Representation Interchange Standards" proposal.

7.1 Start of Work Conference

The start of work conference was held in Alexandria, VA at the IDA offices on 8/12/2002. ICT's conference notes were delivered by email to John Tyler and Joe Toth as well as the other participants on 8/13/2002. A copy of ICT's presentation from the start of work conference can be found on the CD included with this final report.

7.2 Laboratory Design Document

The laboratory design document was delivered by email to Joe Toth on 10/15/2002. An updated version of this laboratory design document is included in this report as Section 3.0.

7.3 Demonstration

The demonstration scenario was demonstrated at the Final Project Review meeting at the IDA offices in Alexandria, VA on 7/16/2003. A slightly updated version of this demonstration scenario can be found on the CD included with this final report or on the FTP site. The demonstration shows three HBM models (Soar, AI.Implant, PMFServ) operating in the same environment through very similar interfaces in a scenario motivated by Black Hawk Down.

7.4 Final Report

This final report will be delivered in draft form by email on 9/26/2003 with electronic copies to be mailed to the program manager at the same time.

Appendix A: Infiltration Mod

Infiltration is a squad-based, mission-oriented add-on that utilizes modern-day weapons (M16-A2), equipment (Kevlar Helmets) and tactics (strafing around corners). Below are the suite of weapon models currently instantiated in *Infiltration*. These are the types that may be used within the demonstration.

PISTOLS
- Beretta M9
- FN Five-seven
- H&K MK 23 SOCOM
- Desert Eagle Mark XIX
- Walther P99

SUBMACHINE GUNS
- Arsenal AKMSU
- FN P90
- H&K MP5/40A3
- H&K MP5K PDW
- H&K MP5N "Navy"
- IMI Mini Uzi

ASSAULT RIFLES
- Colt M4A1 Carbine
- Colt M16-A2
- Giat Famas G2
- H&K G11
- H&K G36C
- SIG SG 551-SWAT

SNIPERRIFLES
- H&K PSG-1
- Robar RC50

SHOTGUNS
- Benelli M1 S90
- Remington 870

MACHINE GUNS
- FN M249 SAW

EXPLOSIVES
- H&K HK69-A1
- M67 Frag Grenade
- M18A1 Claymore

- Smoke Flare
- Signal Flare

Appendix B: Additions/Modifications to Soar's MOUTbot UnrealScript

MOUTInterface.uc

- Added ProcessStrafeTo function (and made required changes to ReportCommand event)
- Made a minor change to ProcessMoveTo, but this change actually has no impact on the system since a) the BHBots always move with strafing and b) it is in a section of code which should never get executed. For these reasons, this change can be ignored.

SoarMOUTBot.uc

- Modified TeamMessage event to send all messages to all bots (so, anything the user sends with the Say command will be received by the bots). This is a bit of a hack, and really, broadcast or something should have been used. If we continue working on this in the future we should revise this, but for now it works.
- Modified "Walking" and "Thrusting" state code to change "Groundspeed". For walking, speed is controlled via a speed parameter in the MoveTo native function (this is a UT native function, not something we wrote). The StrafeTo native function (called from the Thrusting state) does not have a speed parameter, so the speed needs to be changed by changing the Groundspeed directly. We change the Groundspeed to the desired speed in the Strafing state code and change it back to the default in the Walking state code. As a side note, it turns out that the speed parameter of the MoveTo function was misunderstood (it's not what you think) and we will probably change the walking code to look like the strafing code at some point.
- The "TweenToWalking" and "FinishAnim" function calls in the "Walking" and "Thrusting" states were commented out. For reasons that are not well understood, these were preventing bots from moving to a new location until after the user had stopped moving (so they could not maintain formation while the user was in motion).

SpawnBHScenario.uc, BHScenario1.uc

These classes were added to make testing easier. Spawning an instance of BHScenario1 simply created some MOUTBots at specified locations in our test level (not the BHD level) so we could consistently and easily test some situations. These classes are not necessary for the Blackhawk scenario.

SoarBHBot0-3.uc, SoarAPIBHBot0-3.uc

These are the SIO and API versions of the BHBots, respectively. These classes merely specify the UnrealTournament team that the bots are on (red, in this case), their names (BHBot0-3) and the Soar files to load them from (BHBot0-3.soar). For the SIO versions, they also specify the IP addresses (all local machine) and the port numbers to connect to (7605-7608).

SoarMOUTBot1-4.uc, SoarAPIMOUTBot1-4.uc
These are the SIO and API versions of the MOUTBots, respectively. We used these for testing only – they are not required for the Blackhawk scenario. The only significant difference is that we put them on the blue team so that the game didn't end when we killed one (every time someone is killed the game checks to see if there is only one team left; if so, the level ends).

SoarGame Changes
SoarInterface.uc

- Added bDebugDestination variable and associated code in the Tick function to mark where bots were trying to move to. This was very valuable during debugging. It can be turned on or off by setting the variable value in the defaultproperties section. This change is not absolutely necessary for the Blackhawk scenario, but we highly recommend it for debugging purposes. In the future it would be nice to be able to turn this on and off at runtime.

Appendix C: Monthly Reports

March 2003

March Progress Report
DMSO Human Behavior Representation Interchange Standards Laboratory
4/17/2003

Throughout March, we focused primarily on laying out a roadmap leading up to the final demonstration at the end of July. Based on conversations with Dr. Michael Young and Dr. John Tyler we adjusted our task ordering to put more emphasis on developing a candidate HBM interface standard and less emphasis on the integration of the three HBM components with each other. The Interim Progress Report (to follow shortly) describes our progress in interfacing HBM modules with Unreal Tournament in more detail.

HBM component interfacing: The AI.Implant interface to Unreal Tournament was improved in a number of ways.

- Work began on upgrading AI.Implant to the newly released version (version 1.6), which has significant path-planning improvements from earlier distributions
- Continued AI.Implant integration into Unreal Tournament—this includes representing UT static objects within the AI.Implant Object Framework

In addition the interfacing of Performance Moderator Functions in collaboration with the UPenn group got underway. Meetings with UPenn yielded a specification for what weapon sets/models, character meshes/animation, and PMFs will actually be used for the scenario. In addition the AI.Implant interface with resources for writing native functions within UT specific to the HBM was delivered to UPenn to act as a model for their PMF interfacing efforts. Basing the PMF interface off the AI.Implant interface will ensure that all three HBM interfaces can be combined into a single candidate HBM interface in the next few months. Finally, the AI.Implant integration with Unreal Tournament was delivered to Joe Toth to prepare for the evaluation study.

Work on the candidate common HBM interface commenced in March. An initial design for the data types and process flow was developed and work began on bringing the Soar and AI.Implant interfaces with Unreal Tournament into line with this design.

Documentation: Work on fully documenting the Soar interface, AI.Implant interface, and Unreal Tournament interface continued. Fully understanding and documenting these various interfaces will form the basis for the design of the candidate common HBM interface. In addition an Interim Progress Report will be finalized and delivered in the next few days. Also, work commenced on the specification of the art assets needed for the final demonstration scenario.

Laboratory Setup: Final hardware orders were placed and the two machines ordered last month were set up.

Game Developer's Conference: Dr. van Lent attended the 2003 Game Developer's Conference in San Jose, CA. While at the conference Dr. van Lent met with Dr. Paul Kruszewski, lead developers of AI.Implant, and representatives of Epic Software, developers of Unreal Tournament. In addition Dr. van Lent participated in the International Game Developer's Association Academic Summit which is the primary venue for the discussion of how game developers and academic researchers can improve collaboration. Dr. van Lent presented a summary of ICT research projects involving the game development community.

Scenario Development: Based on the input of the PMF team and the game development company Quicksilver Software, the Black Hawk Down scenario continued to be revised and improved.

Meetings:

- Weekly meeting – 3/3
- Weekly meeting—3/12
- UPenn teleconference —3/17
- Budget projection meeting – 3/18
- Weekly meeting—3/20
- Soar teleconference —3/26
- UPenn teleconference – 3/31

April 2003

April Progress Report
DMSO Human Behavior Representation Interchange Standards Laboratory
5/16/2003

Throughout April, we enhanced the AI.Implant (AII) interface and worked with Quicksilver Software to define the Unreal Tournament (UT) artwork that is to be developed for the new MOUT scenario (Mogadishu). This scenario will be used by ICT and UPenn to demonstrate the various HBM components in a variety of roles (friendly soldier, enemy asymmetric, neutral crowd, hostile crowd). The AI.Implant interface was enhanced to support instantiating a waypoint network within the AI.Implant Object Model (instead of the single vertex positions that had previously guided bot movement and navigation). This would not only make movement throughout the level more precise and controllable, but proved to be a much more efficient approach from the AI.Implant Object Model perspective.

During the month of April ICT and Quicksilver Software planned the development of the art assets necessary to support the demonstration scenario. During this planning a study was conducted of the strengths and weaknesses of both the original Unreal Tournament engine and the newer Unreal Tournament 2003 engine. Although the Unreal Tournament 2003 engine has much better graphics support and is more manageable it does not include "native function" support which allows the mod community to integrate external software applications (such as HBM architectures). This "native function" support is the main reason the original Unreal Tournament engine is so widely used by the research community. Since this discovery Dr. van Lent has contact the President of Epic Games (developer of Unreal Tournament) to discuss this weakness of the new engine. Epic Games seems open to the possibility of a "research-only" license for the game engine which would allow the HBM research community to obtain the necessary object code to integrate external software at little or no cost. Discussions with Epic Games are on-going. For the scope of the current project we will use the original Unreal Tournament engine.

HBM component interfacing: The AI.Implant interface to Unreal Tournament was improved in a number of ways.

- Completed and tested upgrade to AI.Implant v1.6
- Began laying out the waypoint network within UT and AII—a shift from standard position coordinates to a waypoint network was determined to be more efficient and practical

In addition the full HBM/Unreal Tournament package was checked into ICT's new SourceForge project management system which includes a Concurrent Version management System (CVS). The HBM/Unreal Tournament package including the Soar and AI.Implant interfaces was made available to the UPenn team to help them integrate PMFServe through our standard HBM interface. Using the example software and documentation we've developed they were able to integrate an initial PMFServe bot in approximately two weeks.

Documentation: All the documentation developed to assist future efforts to interface HBM components to industry game engines was posted on the internal SourceForge server and made available to the UPenn team. In addition, the scenario description document was updated to reflect the art asset development. Finally, the previously developed project schedule was updated.

Laboratory Setup: Final hardware is in place and all software has been installed. This includes the introduction of a new SourceForge project management system including a Concurrent Version management System (CVS) server.

Scenario Development: Based on the input of the PMF team and the game development company Quicksilver Software, the Black Hawk Down scenario was finalized and the final art asset production statement of work was completed. The production of the art assets is set to commence May 2nd.

Meetings:

- 4/2 - Project Status meeting with Randy Hill, Ryan McAlinden and Michael van Lent
- 4/3 – Meeting with Quicksilver to discuss art asset creation
- 4/4 – Weekly project meeting
- 4/7 - Telecon with QSI
- 4/11 – Weekly project meeting including Telecon with UPenn

- 4/14-4/15 – DMSO workshop at ICT
- 4/15 - Telecon with UMichigan
- 4/17 – Weekly project meeting
- 4/18 - Meeting with QSI
- 4/25 – Weekly project meeting including Telecon with UPenn

May 2003

May Progress Report
DMSO Human Behavior Representation Interchange Standards Laboratory
6/20/2003

During May the Human Behavior Model (HBM) interface to Unreal Tournament was mostly completed and the ICT team's efforts started to shift towards developing behaviors and models for the demonstration scenario. With AI.Implant working robustly through the HBM interface work began on developing the AI.Implant waypoints in the demonstration environment. Work started on additional Soar behaviors necessary for the demonstration, starting with a follow behavior. Also the UPenn team showed an early demonstration of the Performance Moderator Function server (PMFserv) working through the HBM interface.

Art asset production for the demonstration scenario was also under way with Quicksilver producing 2 software drops that included:

- Textured terrain data (perimeter walls, blocky building geometry)
- Static Obstacles
- Basic set of animations for scenario characters

This new MOUT environment replaced the existing AII/Soarbot MOUT Environment and will serve as the backdrop for the final HBM demonstration.

The AII interface was enhanced to support a waypoint network within this new level, which consisted of defining path nodes within Unreal Tournament and replicating them within the AII Object Model. Once in the Object Model, network creation began that would eventually navigate the character through the streets of the level.

The team also began development of the "follow operator" within Soar that will allow 3 subordinate Rangers to flock around the user as he advances through the streets. Conversations with the University of Michigan took place to better understand what needed to be done for such an operator implementation.

Discussions also took place with the University of Pennsylvania's PMF group to determine a precise role for their emotion-based software in the BHD scenario. It was concluded that they would produce a group of civilian characters and a single enemy combatant that is controlled by PMFServ.

HBM component interfacing:
AI.Implant:

- Completed final path node and waypoint network declaration in the existing MOUT environment
- Began implementing new path nodes within the Object Model based on the Mogadishu level
- Researched potential automation of UT map data into AII through parsing of t3d file
    - Created a lightweight java parsing program
    - T3d's file format found to be too raw to be useful
- Began logging vertex information for future import into AII

Soar:

- Began design and development of the follow operator that would control movement of 3 subordinate Rangers in UT

General:

- Prepared an AII demonstration for the BRIMS conference: set up a scenario that included an autonomously AII-controlled bot competing against the User, Soarbots, and UT Bots in a DeathMatch mode
- Moved all HBM software over to a clean UT (Infiltration) release; included recompilation of the Soar and AII Interfaces and UnrealScript classes

Documentation:

- Updated Project schedule was produced, along with modified versions of the scenario script and asset documents

Laboratory Setup: Setup of the DMSO laboratory was completed in April.

Scenario Development:

- Coordinated the art asset production with Quicksilver software including the delivery of first two delivery milestones.
- Updated UT Black Hawk Down map with temporary Path Nodes.

Travel & Conferences:

- While on the East Coast for other purposes Ryan McAlinden spent a few hours at IDA delivering software and documentation to Jozsef Toth in preparation for Dr. Toth's talk at the BRIMS conference.
- Dr. van Lent traveled to the BRIMS conference in Mesa, AZ to participate in a panel discussion and attend research presentations.

Meetings:

- 5/2—Weekly Meeting; teleconference with UPenn
- 5/9—Trip to IDA to deliver software and documentation to Jozsef Toth for the BRIMS Conference

- 5/16—Weekly Meeting
- 5/19 – Teleconference with BioGraphics Technologies (AI.Implant)
- 5/27—Weekly Meeting; teleconference with Michigan (Soar Group)
- 5/30—Teleconference with UPenn

June 2003

June Progress Report
DMSO Human Behavior Representation Interchange Standards Laboratory
7/21/2003

During June the Human Behavior Interchange Standards project worked primarily on the demonstration scenario and final tasks. Dr. van Lent and Ryan McAlinden also traveled to Ann Arbor, MI to attend the Soar Workshop and present a talk and initial demonstration. This demonstration showed, for the first time, three Human Behavior Models (HBMs) working in a single environment through a single interface. The talk and demo were well received and multiple groups (U of Michigan, CMU, ISLE) expressed an interest in future collaboration.

Work on the AI.Implant HBM included placing barriers in the AI.Implant spatial representation model corresponding to the buildings in the Unreal Tournament environment. In addition, a flocking model built into AI.Implant was tested as a means for controlling Unreal characters. Work on the Soar HBM included developing new Soar operators to model the behaviors required for the demonstration scenario and extending the Soar/Unreal Tournament interface to support these behaviors. Work on the PMF HBM consisted mainly of supporting the development efforts of the UPenn team as they integrated PMFServ with Unreal Tournament.

In addition, the team oversaw art asset production for the demonstration scenario underdevelopment at Quicksilver producing 1 software drop that included:

- Updated textured terrain data
- New character models

HBM component interfacing:

AI.Implant:

- Created a rough Barrier approximation of the UT level for AI.I
  - Updated as necessary with new level drops
- Implemented a new waypoint network inside AI.I
  - Auto edge generation for the waypoint network, based off of existing barriers
- Tested a prototype SeekToViaNetwork behavior for Leader movement
  - Tweaked animation
- Tested a prototype FlockWith behavior for SquadMember movement
  - Tweaked motion/starting locations
- Extended Maneuver.dlls logging

- Extended Maneuvers UScript logging capabilities
    - Ability to log to a specified file instead of default
- Extended AI.I/UT interface
    - SetAgentPosition function
    - SeekTo function (deprecated already)
    - FlockWith function (deprecated already)
- Changes and recompilation of the native functions as a result of ID conflicts that existed between AI.Implant and PMFserv
- Upgraded to AI.Implant release v1.6.1

Soar:

- Wrote Soar operators and productions to support follow and orient behaviors that allow a user in Unreal Tournament to spawn four subordinate Soarbots that oriented around him in a box formation.
- Modify the Soar Interface (on the UnrealScript side) to allow the Soarbots to move and orient themselves correctly as related to the User

General:

- Integrated Soar HBM, AI.Implant HBM, and PMFServ HBM so all three bots could interact in the same environment.
- Extended Maneuver.dlls logging
- Extended Maneuvers UScript logging capabilities
    - Ability to log to a specified file instead of default
- Researched ways around Summoning the bots at game time
    - Able to have pre created bots, problems with spawning their inventory
    - Work around – bots Spawned inside AIISetup which must be summoned (1 Summon instead of 4)
- Developed UScript method of bot self initialization
- Developed C++ method of bot self initialization
- Implemented UT's forcing of bot locations every tick

Documentation:

- Commenced work on the final report.

Laboratory Setup: Setup of the DMSO laboratory was completed in April.

Scenario Development:

- Coordinated the art asset production with Quicksilver software including the delivery of third delivery milestones.
- Modification of the Mogadishu level to include a preliminary set of path nodes used by a variety of the bots for their internal terrain representation

Travel & Conferences:

- Attended the 2003 Soar Workshop in Ann Arbor, MI. Presented a talk titled "Human Behavior Models and Unreal Tournament" and a demonstration.

Meetings:

- 6/4 – Teleconference with Quicksilver
- 6/5 – Weekly meeting
- 6/6—Telecon with UPenn
- 6/11—Telecon with UPenn
- 6/20 – Weekly meeting
- 6/23—Telecon with UPenn
- 6/23 – Status report meeting with Ryan McAlinden
- 6/24-6/27—Soar Workshop